NAVAL POSTGRADUATE SCHOOL
Monterey California

THESIS

RENDERING LARGE-SCALE TERRAIN MODELS IN 3D AND POSITIONING OBJECTS IN RELATION TO 3D TERRAIN

by

Brian Edward Hittner

December 2003

Thesis Advisor:	Don Brutzman
Second Reader:	Curt Blais

This thesis done in cooperation with the MOVES Institute.
Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Rendering Large-Scale Terrain Models and Positioning Objects in Relation to 3D Terrain			**5. FUNDING NUMBERS**
6. AUTHOR Brian E. Hittner			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE** A

13. ABSTRACT

Computer generated 3D graphics have been commonplace in computing since the early 1990's. However, most 3D scenes have focused on relatively small areas such as rooms or buildings. Rendering large scale landscapes based on 3D geometry generally did not occur because the scenes generated tended to use up too much system memory and overburden 3D graphics cards with too many polygons. However, there are applications where the terrain is critical and needs to be rendered properly such as cartography and military simulation. This thesis is focused on methods of rendering terrain for such applications.

The data used to build terrain geometry typically comes from elevation postings taken from surveys of the terrain. This thesis does not focus on collecting this data nor does it compare various sources of terrain data. Instead, this thesis is about taking elevation data, producing a rendered 3D scene, and placing objects within the scene relative to the terrain. Having these capabilities makes many military and cartographic applications possible. Some military applications include displaying the results of computer simulations in 3D, planning operations using a 3D landscape, and rehearsing operations in 3D. The military does have some tools that can be used today for these actions, but the tools are typically proprietary and expensive. This thesis is focused on using and extending open source tools for 3D terrain rendering. The result is tools that can be freely used, studied, and expanded by anyone without licensing costs.

14. SUBJECT TERMS Terrain, X3D, Virtual Reality Modeling Language, Digital Terrain Elevation Data			**15. NUMBER OF PAGES** 133
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

This thesis done in cooperation with the MOVES Institute.
Approved for public release; distribution is unlimited

RENDERING LARGE-SCALE TERRAIN MODELS AND POSITIONING OBJECTS IN RELATION TO 3D TERRAIN

Brian E. Hittner
Captain, United State Army
B.S., United States Military Academy, 1994

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS AND SIMULATION

from the

NAVAL POSTGRADUATE SCHOOL
December 2003

Author: Brian Edward Hittner

Approved by: Don Brutzman
Thesis Advisor

Curtis Blais
Thesis Co-Advisor

Rudolph P. Darken
Chair, MOVES Curriculum Committee

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Computer-generated 3D graphics have been commonplace in computing since the early 1990's. However, most 3D scenes have focused on relatively small areas such as rooms or buildings. Rendering large scale landscapes based on 3D geometry generally did not occur because the scenes generated tended to use up too much system memory and overburden 3D graphics cards with too many polygons. However, there are applications where the terrain is critical and needs to be rendered properly such as cartography and military simulation. This thesis is focused on methods of rendering terrain for such applications.

The data used to build terrain geometry typically comes from elevation postings taken from surveys of the terrain. This thesis does not focus on collecting this data nor does it compare various sources of terrain data. Instead, this thesis is about taking elevation data, producing a rendered 3D scene, and placing objects within the scene relative to the terrain. Having these capabilities makes many military and cartographic applications possible. Some military applications include displaying the results of computer simulations in 3D, planning operations using a 3D landscape, and rehearsing coordinated operations in 3D. The military does have some tools that can be used today for these actions, but such tools are typically proprietary, not interoperable and expensive. This thesis is focused on using and extending open source tools for 3D terrain rendering. The result is tools that can be freely used, studied, and expanded by anyone without licensing costs.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

This thesis would not have been possible without the work of the SAVAGE group at the Naval Postgraduate School. The SAVAGE group has been aggressively pursuing graphics technologies using X3D and VRML. The work of this group provided the foundation for this thesis. In particular, CPT James Neushul's DTED server software that provided the terrain files that were the basis for the GeoTerrainGrid node, which is where the majority of the work was done for this thesis. Likewise, the SAVAGE group provided the library of military vehicles that the tanks used in the example files came from.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

There are many military applications that depend upon having an accurate representation of terrain including simulating battles, planning operations, and rehearsing technical maneuvers. Simulating battles requires determining if opposing forces can see each other and engage each other. This requires line of sight calculations based on the positions of the opposing forces and the terrain separating them. Planning operations requires knowing the steepness of terrain to determine where units can move and line of sight calculations (again) to determine engagement areas. Rehearsing military operations requires as detailed a visual representation of the terrain as possible so that the participants in the rehearsal can recognize the significant terrain features later when executing the operations. All of these applications depend upon accurate renderings of terrain or accurate analysis of the terrain. The underlying source of data for the terrain is typically a two dimensional (2D) array of height values called a height field. The problems with using computers as analysis tools for terrain begin with these height fields. This thesis looks at six specific problems with using terrain based on height fields: convert the height field into a renderable object, calculating the elevation of an arbitrary position, calculating the orientation that an object will have if resting on the terrain at an arbitrary point, calculating line of sight (LOS) based on height fields, calculating LOS based on curvature of the earth, and reducing the polygon count of rendered height-field terrain to improve the run-time performance of the rendering engine.

B. OVERVIEW

This thesis starts with the Virtual Reality Modeling Language (VRML) as a rendering engine. VRML alone does not support geographic coordinates, though, so the GeoVRML extension to VRML is used. GeoVRML allows a standard web browser such as Internet Explorer to render graphics that are specified in GeoDetic Coordinates (GDC), Universal Transverse Mercator coordinates (UTM), or GeoCentric Coordinates (GCC). Geodetic Coordinates are more commonly known as latitude and longitude values. Universal Transverse Mercator coordinates allow specifying locations as a northing value

(meters north-south) and an easting value (meters east-west) referenced within one of sixty world zones. Both of these coordinate systems are basically 2D systems that are mapped to the varying surface of the earth.

GeoCentric Coordinates are part of a true three-dimensional (3D) coordinate system. A coordinate specified in GCC is a triplet containing an X, Y, and Z value. Converting between GDC and UTM is fairly straightforward since latitude and northing values are basically just a measurement of distance from a boundary, i.e. either the equator or a zone boundary. Converting between longitude and easting values is a little more complicated because the distance between successive lines of longitude changes with latitude. Lastly, the height values do not change when going from GDC to UTM as both are meters above sea level. Converting to and from GCC is more complicated, though. A change in latitude does not map to a change in one variable of a GCC triplet. Instead, a change in latitude can cause all three values in the GCC triplet to change. Changing longitude or elevation has similar complex effects. Fortunately, most users do not need to work frequently with GCC coordinates as they are primarily needed by the computer for rendering.

This thesis deals with all three of the above-mentioned coordinate systems: UTM, GDC and GCC. However, the transformations between the systems are left to the GeoTransform portion of the GeoVRML code. Why transformations are needed is now addressed, but the math behind doing the transformations is not yet discussed. The computation of these transformations is significant. If these transformations are computed or converted incorrectly, then objects and terrain will be rendered in the wrong locations. In military applications, such problems can lead to disaster. To address this concern, the geospatial software used in this thesis is designed so that these transformations are all done in an isolated package that can be validated as part of another study or swapped out with a package that has been validated. The GeoTransform package that is being used is freely available to download, view, and use. The end result is that all of the code built for and used in this thesis is available freely on the Internet without any significant restrictions on viewing or re-using source code.

Each of the six problems this thesis deals with depend on some or all of the coordinate systems mentioned above. The first problem, translating height fields into 3D rendered objects, depends upon translating GDC and UTM coordinates into GCC coordinates and then connecting the coordinates in the proper sequence. Calculating the elevation of arbitrary positions depends upon defining a plane based on GDC or UTM coordinates along with the height field to calculate the elevation of some point on that plane. Calculating orientation at arbitrary positions is similar except that the plane must be defined in GCC coordinates and the normal calculated along with a rotation vector to coincide with that normal. The fourth problem, calculating LOS based on height fields requires using GDC or UTM coordinates to determine distances between elevation postings so that angles representing possible lines of vision can be calculated and compared. Calculating the LOS based on curvature of the earth is similar, but the height field is not necessary since the horizon is far more significant. Finally, reducing the number of polygons in rendered terrain also involves calculating distances between points and determining the minimum number of polygons needed to render the terrain. Each of these problems are covered in detail in Chapter III and IV.

C. MOTIVATION

Developing this thesis furthers military computer simulations. There are several military simulations that are already rendered in 3D such as flight simulators and tank gunnery trainers, but the bulk of current maneuver-training simulations are rendered from a 2D top-down perspective. The techniques developed in this thesis are a foundation upon which existing 2D simulations can be displayed in 3D or new maneuver simulations can be built in 3D. These techniques can also be used in real-time applications such as planning military operations and conducting visual rehearsals. However, since the underlying GeoTransform package has not been validated, care should be taken when using the code developed here in real-world applications. Either the GeoTransform package must be validated or replaced by another validated package before even considering using the work in this thesis for real-world military applications. If validation does not occur, then any decisions about battle positions, artillery targets, or any other decision regarding munitions effects or determining locations of cover should be double checked with real-world reconnaissance efforts.

Of course, products from this thesis can also benefit real-world military operations in addition to simulations. Part of planning military operations is Terrain Visualization. Army Field Manual 3-34-230 Topographic Operations [FM 3-34-230 p. 1-3] defines Terrain Visualization as the process through which a commander sees the terrain and understands its impact on the operation in which a military unit is involved. Viewing the terrain in true 3D with forces arrayed on that terrain properly can assist the commander in this task. Currently, most units still do most if not all terrain analysis using traditional 2D paper maps. Here is a diagram from FM 3-34-230 depicting Terrain Visualization.

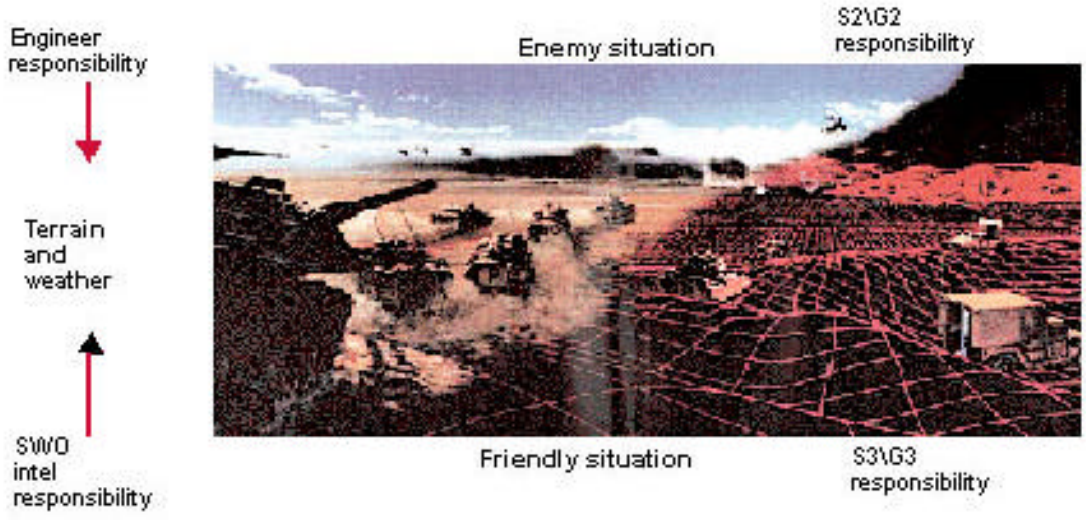

Figure 1. Terrain visualization concepts [Army FM 3-34-230].

D. THESIS ORGANIZATION

This thesis is organized in four chapters. This first chapter provides an introduction to the topics being discussed including the six major areas of study. This thesis is aimed at improving the rendering of 3D terrain based upon height fields and placing objects on that terrain.

The second chapter focuses on the background material studied to complete this work. Three geographic coordinate systems used to identify locations on a global scale are studied: UTM, GDC, and GCC. Then, Digital Terrain Elevation Data (DTED), a commonly used format for encoding data for height fields, is examined. Next, the Extensible Markup Language (XML) is briefly discussed as a tool for extracting height values from DTED files and organizing them into objects that can be rendered. The second chapter continues with a discussion of the Virtual Reality Modeling Language (VRML) which is used as the rendering engine in this thesis. The engine contains support for indexed face sets which are critical to this thesis. An extension to VRML called GeoVRML is covered because it is needed for mapping global coordinates in VRML. This discussion concludes with an introduction to X3D, the next version of VRML, and the corresponding X3D GeoSpatial component.

Chapter III describes the work done in this thesis. This works is divided into three sections each of which describes a program that handles part of the task of rendering terrain and placing objects on that terrain in 3D. These subjects are the GeoManager program which allows communication between terrain and objects placed on that terrain, the GeoTerrainGrid program which renders terrain and does all calculation of elevation and slope of terrain, and the GeoLocation3 program which objects use to place themselves on terrain automatically. All of the code is written in Java so that it can be used as VRML script files.

Chapter IV contains conclusions and recommendations for future work. Under the conclusions, the current abilities of the code built in this thesis are reviewed along with the limitations. The future works section lists five specific areas where this thesis can be extended. The topics are discussed in order from the simplest to implement to the most difficult.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND AND RELATED WORK

A. INTRODUCTION

This chapter begins by introducing three different coordinate systems for locating positions on a global scale. The first two (UTM and GDC) are common for real-world navigational and cartographic uses while the third (GCC) is a computer specific system for rendering geography. The next subject is Digital Terrain Elevation Data or DTED which is a common and readily available military standard for storing data about terrain. The Extensible Markup Language (XML) is then introduced as a tool to help work with DTED. Finally, X3D and the Virtual Reality Modeling Language (VRML), the rendering engines that the code in this thesis uses, are described.

B. GEOGRAPHIC COORDINATE SYSTEMS

1. Latitude and Longitude

Latitude and longitude are a spherically based mapping system for the earth. Imagine a coordinate axis being placed inside the earth with the origin at the earth's center. The x coordinate axis extends from the origin through the point where the equator (latitude 0°) and the prime meridian (longitude 0°) meet. The positive z coordinate axis extends from the origin through the North Pole. Finally, the positive y axis extends out from the origin so that it is perpendicular to the x-z plane and intersects the surface of the earth in the eastern hemisphere (somewhere in the Indian Ocean). A drawing of this coordinate axis system can be verified with the right-hand rule. Imagine grabbing the z coordinate axis with the right hand such that the fingers curl around from the x-axis toward the y axis. If the thumb is pointing up along the z coordinate axis, then the coordinate system is properly placed. The following diagram (Figure 2) from the NPS distributed learning module on maps and coordinates depicts this. The online tutorial containing this diagram was created by James R. Clynch [Clynch website] and is located at http://www.oc.nps.navy.mil/oc2902w/c_mtutor/index.html.

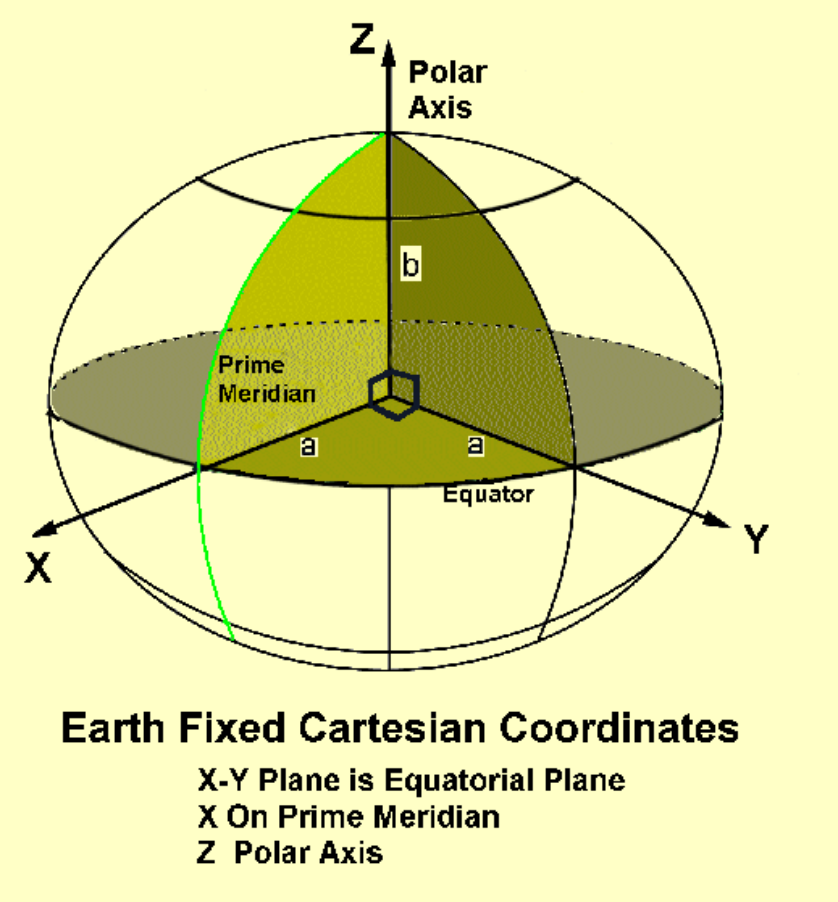

Figure 2. Depiction of earth with Cartesian axis

The latitude component of a latitude and longitude coordinate determines how far north or south from the equator the location belongs. The range is from 0° to 90° and refers to either north or south. Thus, 90° north is the North Pole and 90° south is the South Pole. The equator is simply 0° and could be referred to as either north or south. These degrees are further subdivided by sixtieths into minutes and seconds. One second of latitude is equal to about 30 meters. One degree of latitude is likewise approximately 111 kilometers. These distances are constant all over the surface of the earth. Thus, any specific latitude defines a circle around the earth that is a constant distance from the equator. Since the earth is a sphere, these circles get smaller the closer they are to the poles where they reach a circumference of zero. Sometimes, systems prefer to use negative values for southern hemisphere coordinates and positive values for the northern

hemisphere. This allows the values be stored strictly as numbers without needing an additional character to hold the n or s designation.

The longitude component of a latitude and longitude coordinate determines a semicircle that reaches from the North Pole to the South Pole. Since there is no natural point of reference for these semicircles like there was for the equator, an arbitrary reference point needs to be chosen. The Prime Meridian longitude 0° was chosen as the longitudinal semicircle that travels through the observatory at Greenwich, England. The rest of the earth is divided into 360 slices, each of which is 1° wide. The slices are numbered from 0° to 180° and then designated as east or west. Longitude 180° itself can be designated east or west, and refers to the line of longitude directly opposite the Prime Meridian. Like the latitude values, some systems prefer to simply use a number and designate positive numbers as east and negative numbers as west. The following diagram depicts the earth using latitude and longitude. Once again, this diagram came from [Clynch website].

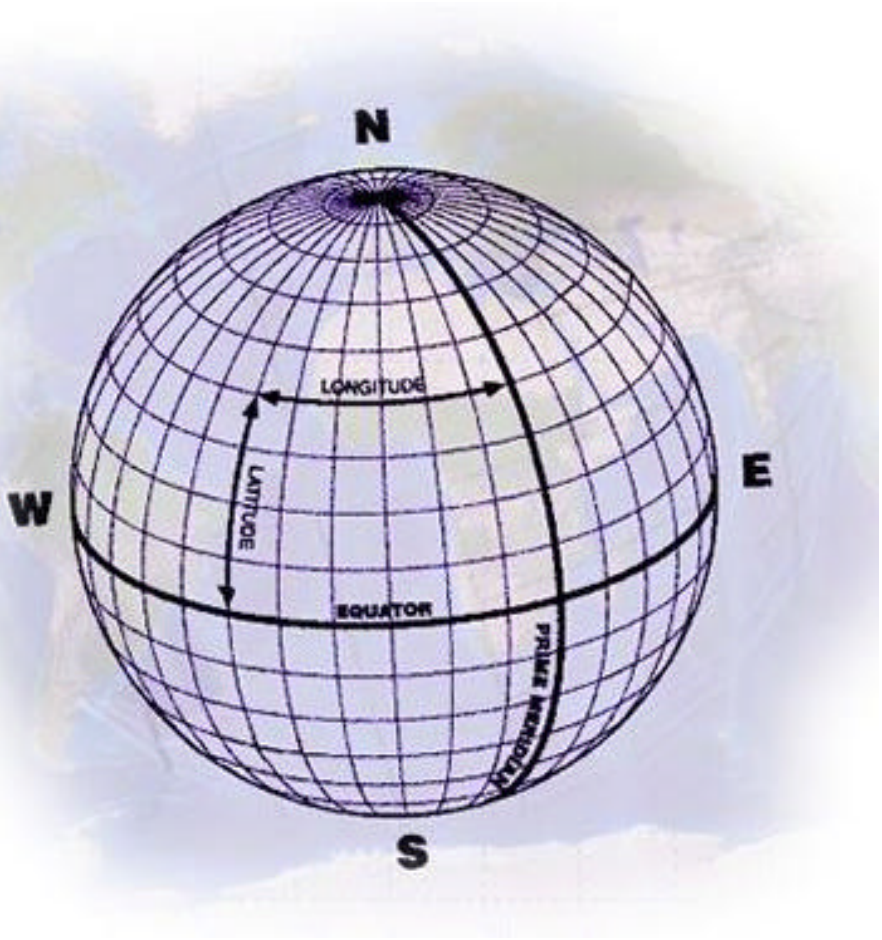

Figure 3. Earth with latitude and longitude system

Latitude and longitude coordinates also frequently contain a height value referred to as the height above sea level. Technically, this is not accurate. The height refers to the height above a model of the earth known as an ellipsoid. There are many ellipsoid models to use, but in this thesis the most common one, WGS84, is used. This ellipsoid is used on most military maps and is the ellipsoid upon which DTED height values are based. Using the height above the ellipsoid is much more intuitive than using the distance from the center of the earth and the resulting numbers are much smaller allowing them to be held as floating point numbers with reasonable accuracy. Floating point numbers take only half as much memory as double precision variables making the files holding large amounts of height data smaller. As for the reasonable accuracy, the importance of that is presented in section E of this chapter.

Together, the latitude, longitude, and elevation (height) values locate a position in 3D relative to the center of the earth. However, the coordinates must be converted to GCC coordinates based on the geoid for a computer to render the point. The details of how this conversion happens are beyond the scope of this thesis. Instead, the GeoTransform package from SRI that is included with GeoVRML is used. GeoVRML is available freely on the web at www.geovrml.org. Using a separate software package for conversions between coordinate systems has three advantages. First, the code for making transformations is all located in one location. Every place where a transform is needed calls the same transformation routines guaranteeing the same results. Second, other programs that work with geographic coordinates can use the same package and will therefore get the same results. This helps make programs work together when analyzing or displaying terrain and objects placed on that terrain. Finally, having all the transformation code located in one package allows the code to be validated separately from the programs that use it. Once the code is validated, users have much greater certainty that the programs that use the validated package are accurate. With military applications, such validation is important. Likewise, if the package is not validated but another transformation package is, then the validated package can replace the package that is not validated with minimal changes to the program. For details about how these transformations are done, the reader is referred to the GeoTransform package developed by SRI [Web 3D Consortium GeoVRML specification http://www.geovrml.org/geotransform/].

Latitude and longitude coordinates are an excellent way to define locations on the earth because they are based on a spherical mapping system. However, latitude and longitude can be difficult to work with when navigating and when running simulations. The primary problem is that moving east and west (i.e. changing longitude) is difficult to compute. If an object is at the equator and moving east at 30 meters per hour, then the object is also moving one second of longitude per hour. Because longitude lines converge as they approach the North of South Pole, an object moving at 30 meters per hour east located at the same latitude as Washington, D.C., travels 1.25 seconds of longitude per hour. The velocity is the same, but the distance between lines of longitude

varies with latitude. Therefore, speed does not map to changes in longitude easily, and distances are hard to determine based on changes in latitude and longitude coordinates. To address these shortcomings, many mapping schemes have been built on distance-based coordinate systems. The most common of these schemes is described in the next section.

2. Universal Transverse Mercator (UTM)

In the Universal Transverse Mercator or UTM geographic coordinate system, the earth is divided into grid zones so that coordinates are specified with northing and easting values. The concept is that the northing and easting values are measured in meters instead of degrees like latitude and longitude. This benefits users who are interested in quickly and easily measuring distances between points. In fact, the distance between two UTM coordinates can be calculated using the Pythagorean Theorem taught in high school algebra. With latitude and longitude, changes in longitude vary in distance as described previously in this thesis. Objects moving at known speeds can be tracked in UTM coordinates easily. The speed can be broken down into an easterly speed and a northerly speed. These speeds are multiplied by time to get distance which is then added to the easting and northing values to get the new coordinate. The drawback to UTM coordinates is that the system is not based on spherical coordinates. Instead, UTM coordinates are simply a two dimensional mapping of the earth. Since the earth is a sphere, some distortion occurs when mapping the curved surface of the earth to the flat surface of UTM coordinates.

The amount of distortion introduced into UTM mappings is limited by mapping sections of the earth to UTM individually. The earth is divided into 60 zones each of which is 6 degrees of longitude wide. Zone 1 starts at 180 degrees west and proceeds east for 6 degrees. These end up being narrow longitudinal zones that are approximately 667 km wide at the equator and narrower when closer to the poles. Inside a zone, the fundamental grid square is 100 km wide by 100 km long. Since the zones are 667 km wide at the equator, this system does have some grid squares that are truncated. At the poles, the system changes even more as the poles actually have special zone numbers and a somewhat different method of mapping the surface. This thesis is not going to describe the UTM system in any greater detail than this. The rendering system used in this thesis

requires that the coordinates contain a northing value that is the distance in meters from the equator (with southern hemisphere values being negative numbers), an easting value that is the distance from the western edge of the current zone, and the zone number (1 to 60) for the zone of the coordinate. Technically, there is also a boolean value labeled northern hemisphere. When true, the coordinate is in the northern hemisphere. This allows the user to specify southern hemisphere locations without using negative numbers. However, the variable can be left true and southern hemispheres coordinates can be specified as negative numbers. The following diagram depicts the earth with the UTM system superimposed on it. This diagram is courtesy of Professor Steven Dutch, Natural and Applied Sciences at the University of Wisconsin, Green Bay. The article containing the diagram and more details about UTM is at [Dutch http://www.uwgb.edu/dutchs/FieldMethods/UTMSystem.htm].

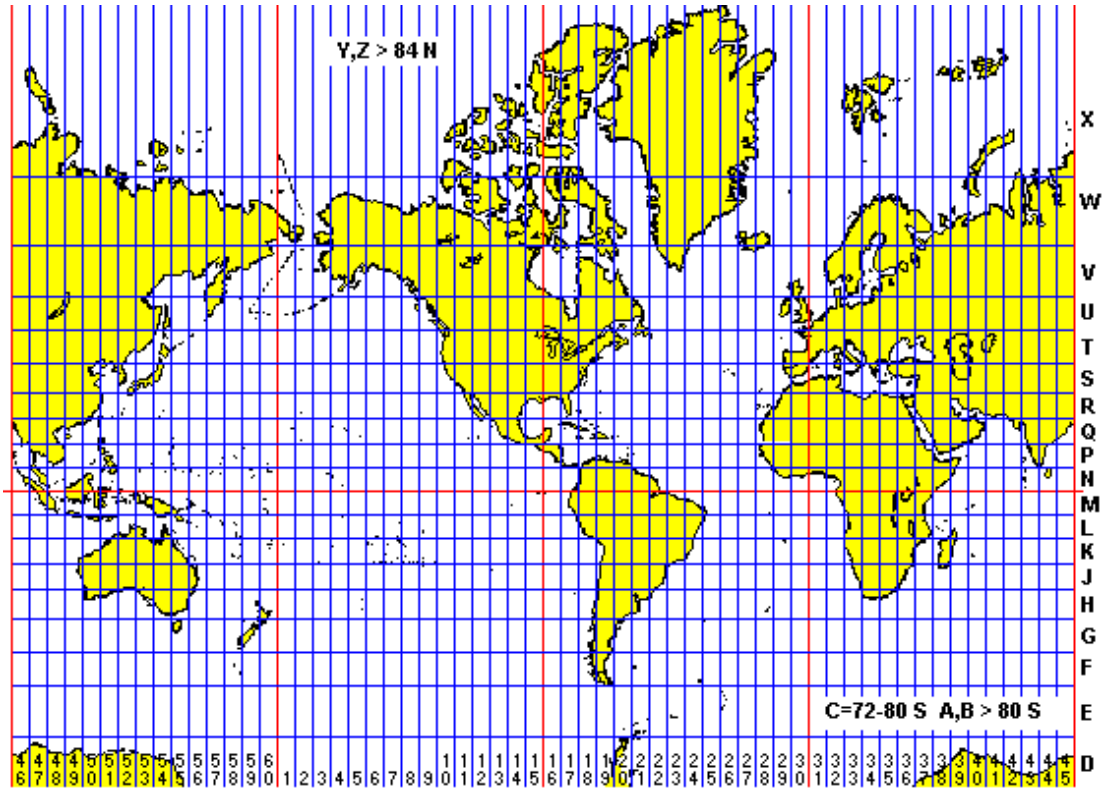

Figure 4. Earth as described by the UTM system.

3. GeoCentric Coordinate System (GCC)

Neither of the two previous coordinate systems can be displayed directly by the 3D rendering system used in this thesis. To display coordinates or anything built from

those coordinates, they must be transformed into a 3D Cartesian coordinate system (GCC). In GCC, each coordinate contains an X, Y, and Z value that together reference a location in 3D space. GCC coordinates are generally only used for display purposes. Tracking units in GCC is even more difficult than using latitude and longitude. For example, consider an object circling the globe while moving east. In latitude and longitude this means simply increasing the longitude being sure to keep the value within the legal values. UTM is a little more complicated because there are 60 zones that the object will pass through as it goes around the world. In GCC coordinates, the X, Y, and Z values will be changing with every move. However, GCC coordinates are useful for calculating the orientation of objects that have already been translated to and placed in GCC space. How GCC coordinates are used will be made clearer later in this thesis.

C. DIGITAL TERRAIN ELEVATION DATA (DTED)

Digital Terrain Elevation Data (DTED) is a dataset created under the direction of the United States Department of Defense (DOD) to help map the earth. Simply put, the National Imagery and Mapping Agency (NIMA) uses various methods to obtain the elevation above sea level of most areas of the planet. The result is a series of matrixes of elevation postings spanning the globe at specific intervals. The lowest level of DTED used in this thesis is DTED level 1 which contains an elevation posting approximately every 90 meters. This thesis also utilizes DTED level 2 which has postings every 30 meters. Higher levels of DTED exist, but levels 1 and 2 are not classified and more readily available than higher levels. DTED does not contain any data about the terrain it covers other than spot elevations. Any other information must be estimated or obtained from other sources. However, DTED elevation postings work well for building 3D models of the terrain they cover. In building these models the landform, slope, and terrain roughness are all approximated and displayed on the computer screen based on the elevation postings that DTED provides. Of course, using higher levels of DTED data gives better approximations and better looking 3D models. However, the amount of data stored and modeled is increased significantly.

DTED data is stored in a standard binary format specified in MIL-PRF-89020A dated 19 April 1996. Basically, the specification details the file structure for individual DTED files that cover small areas of the earth. DTED is commonly distributed on CDs.

However, the earth is large and even at DTED level 1 requires numerous CDs to hold that much data. Just to estimate this, the earth has a radius of approximately 6,300 km. Thus, the surface area is approximately 500 million km^2. At DTED level 1, this requires 50 billion height values for the entire globe. If each height value is stored as a 4 byte single-precision value, then 200 billion bytes are needed. Thus, DTED quickly can reach into hundreds of gigabytes even at level 1. Level 2 has nine times as much data. However, the oceans are always at sea level so DTED typically does not store data for ocean areas – only land masses. This reduces the total amount of data significantly so that the whole world at DTED level 1 can be stored on most modern day hard drives. Trying to display the entire world at DTED level 1 at one time, though, is still beyond the capabilities of today's computers because today's computers do not have the tens or hundreds of gigabytes of RAM required to hold that much data in the computer's memory. Therefore, this thesis focuses on much smaller pieces of terrain so that most computers will be able to render the examples. However, the code presented is capable of handling as much data as the computer can store and display.

D. EXTENSIBLE MARKUP LANGUAGE (XML)

The DTED format is fairly efficient for storing the large amounts of data generated. However, utilizing that data can be difficult due to the low level binary format of DTED. Reading binary files is difficult because an application program has to know the format of the file byte-by-byte in order to reconstruct the higher level data structures such as integers, floating point numbers, and strings. If the application program gets off by one byte, then the program will typically crash or at least produce meaningless data. A more robust system might read a file for the user and break it up into its components based on information contained in the file itself. The user might then simply ask for the dimensions of the data, the location of the data in the world, and the height field values without worrying about how many bytes constitute each height value or location parameter.

The Extensible Markup Language (XML) was designed to meet this need and more. XML is a markup language for data which means that data is stored along with some information that describes the data. The data is enclosed in matching tags that describe the data. For example, a height field of four values could look like this:

<heightField>1000 1002 998 1001</heightField>

This data has an opening height field tag <heightField> and a closing height field tag </heightField> with four height values between them. An XML parser reads this data and stores the four values with the name heightField. If an application asked for the heightField, then the parser would pass the four values to the application. The application does not have to know anything about how the four height values were stored in the file because the XML parser handles all those details. The application's responsibility is to know what to do with the height values. Attributes can also be stored within these element tags that provide more data or provide metadata (data describing the data). There can be tags that give the application information about what DTED level the data is stored in and what coordinates the height data is located at in the world or there could be attributes that store this data. Either system makes the data available to the user. Using XML allows the application writer to focus more on the data and how it is used without worrying about how the data is stored and retrieved.

Of course, XML does more than just store and retrieve data. XML can also be used to validate data files and to transform data files into other formats. Validating files uses a technology called XML Schema, typically stored as XMLSchema Documents (XSD). Basically, an XSD file is created to describe the structure of specific XML documents so the contents can be checked and validated for the proper form. For example, an XSD document could check that all elevation postings are within height field tags and that all the information required is present. The template file can even make sure that data follows specific rules such as integers only or values within a certain range, etc. However, this technology is not directly used in this thesis. It is mentioned because the technology can be useful in future applications that use the techniques and code from this thesis in larger terrain-rendering projects. An application that uses networking to handle terrain between several computers might be a good example.

Another powerful feature of XML is its ability to transform data from one format to another. This technology is called Extensible Stylesheet Language for Transformations (XSLT). Using XSLT, terrain data files can be customized for specific applications. This thesis used one such transformation developed by Captain James

Neushul, a student at the Naval Postgraduate School at the time this thesis was written. CPT Neushul's code reads DTED files, parses them, and constructs renderable scene files written in X3D or VRML, both of which will be discussed later in this chapter. The files produced are scene graphs that when run using an X3D or VRML browser create a geoElevationGrid described in the next section which covers VRML [Neushel 2003]. This thesis uses these files as a starting point, and then investigates how the terrain data is rendered into 3D objects and how objects can be placed on this terrain. The data in these files are the same elevation postings found in the original DTED files; it has simply been transformed into a format that is easier to work with. This way, more effort is spent on rendering issues while avoiding input/output issues involved in parsing binary files.

E. X3D AND THE VIRTUAL REALITY MODELING LANGUAGE (VRML)

A rendering engine is a program that takes a description of objects in 3D and actually draws the pixels on the screen that make a picture of the 3D scene described. The math behind these conversions is complicated and beyond the scope of this thesis. VRML was the first popular 3D rendering language built specifically for the internet. VRML programs are called "world" files and they end with the extension .wrl. When executed, the VRML program renders a 3D scene inside a web browser such as Internet Explorer or Netscape Navigator. Extensible 3D Graphics (X3D) is a new version of VRML. X3D currently has to be transformed through XSLT before viewing as VRML, and native X3D rendering engines are beginning to appear. The specification for X3d is available at www.web3d.org/x3d.

1. VRML Graphics Basics

To display VRML programs, users must download a plug-in for their browser that includes a VRML rendering engine. Several are available free of charge such as the Cortona player at www.parallelgraphics.com/products/cortona. The Web3D Repository has numerous links to browser plug-ins and other useful VRML tools at http://www.web3d.org/vrml/vrml.htm. The VRML language allows the user to build a scene graph that describes a 3D scene. The VRML world is built on a 3D Cartesian coordinate system that follows the right hand rule and has the origin at its center. Initially, the positive X-axis is to the right, the positive Y-axis is up, and the positive Z-axis comes out of the screen. The Figure 5 illustrates this using a coordinate axis scene

found at

http://web.nps.navy.mil/~brutzman/Savage/Tools/Authoring/_pages/page03.html.

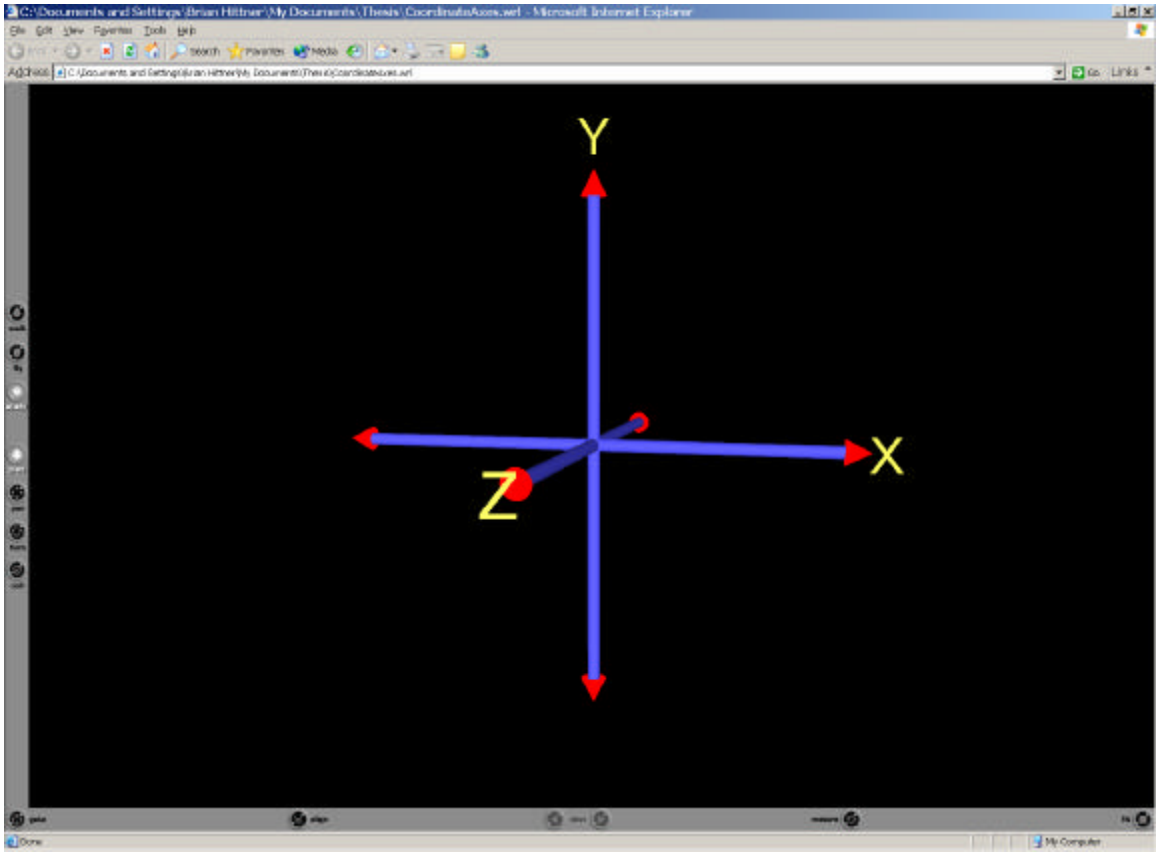

Figure 5. Right-handed coordinate axes showing VRML/X3D coordinate space

Users define shapes out of existing primitives such as spheres and boxes or out of polygons built from coordinates. The coordinates are floating point values or single precision values. VRML also provides ample support for coloring objects, applying textures to objects, and placing lights in the scene. There are position interpolators that move objects around based on key frame positions and timers. Likewise, there are touch sensors and proximity sensors that allow interaction. There are even video and sound objects. The current X3D specification also added keyboard support. In short, the language has most if not all of the components needed to make complicated scenes that have user interaction.

A detailed explanation of how to build 3D scenes in VRML is available in [Ames 1997] that describes every node built into VRML 97 in detail. Another excellent

resource is the website www.vrmlsite.com where numerous articles about VRML are stored. Likewise, information about X3D is available at http://www.web3d.org/x3d.html with many example models and scenes at http://web.nps.navy.mil/~brutzman/Savage/contents.html.

Examples that show the power of X3D/VRML to render detailed objects follows. These objects are true 3D objects that can be viewed from any angle interactively. This thesis develops a method to place objects, such as these examples, on large-scale landscapes using standard geographic coordinates.

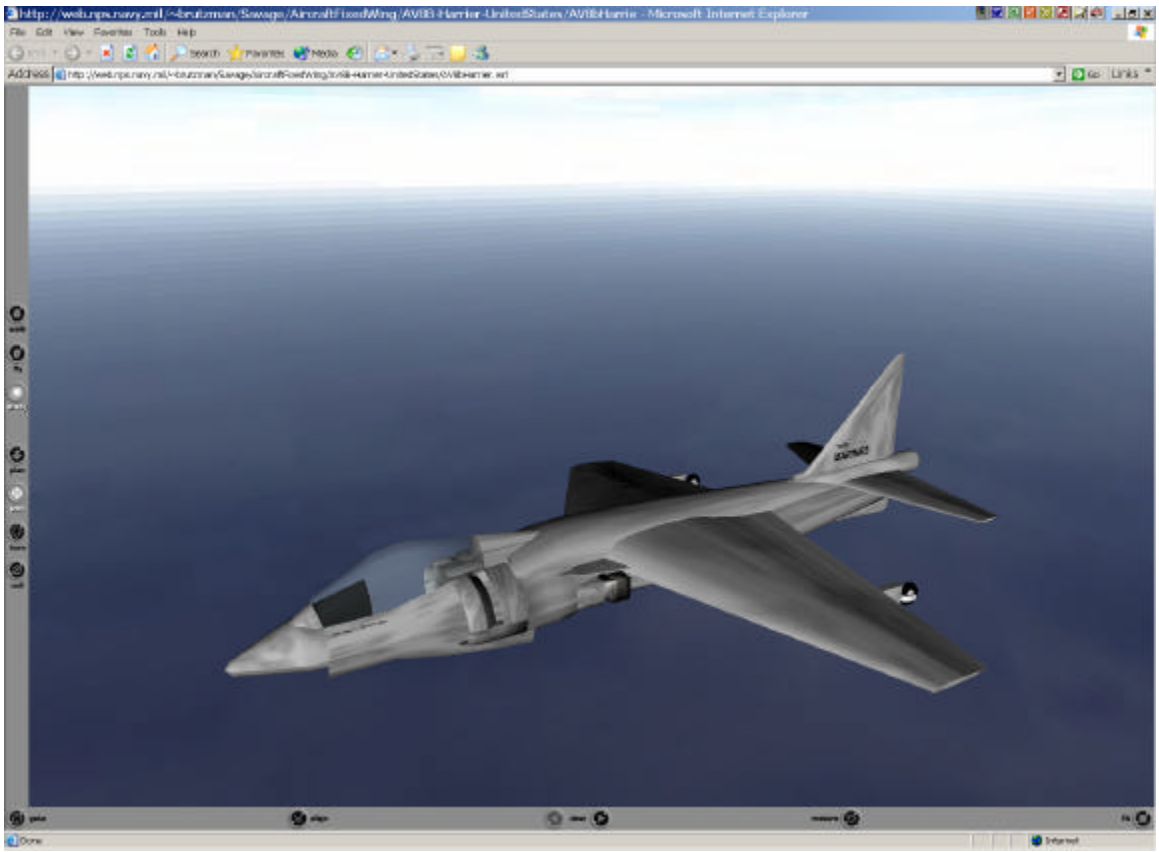

Figure 6. AV-8B Harrier by Miguel Ayala at [http://web.nps.navy.mil/~brutzman/Savage/AircraftFixedWing/AV8B-Harrier-UnitedStates/_pages/page01.html]

Figure 6 is a model of US AV-8B Harrier aircraft. This model can be used in a true 3D military simulation to make the system more visually realistic than the more common 2D simulation systems with 2D icons. Here are some ground-vehicle examples that also come from the SAVAGE collection.

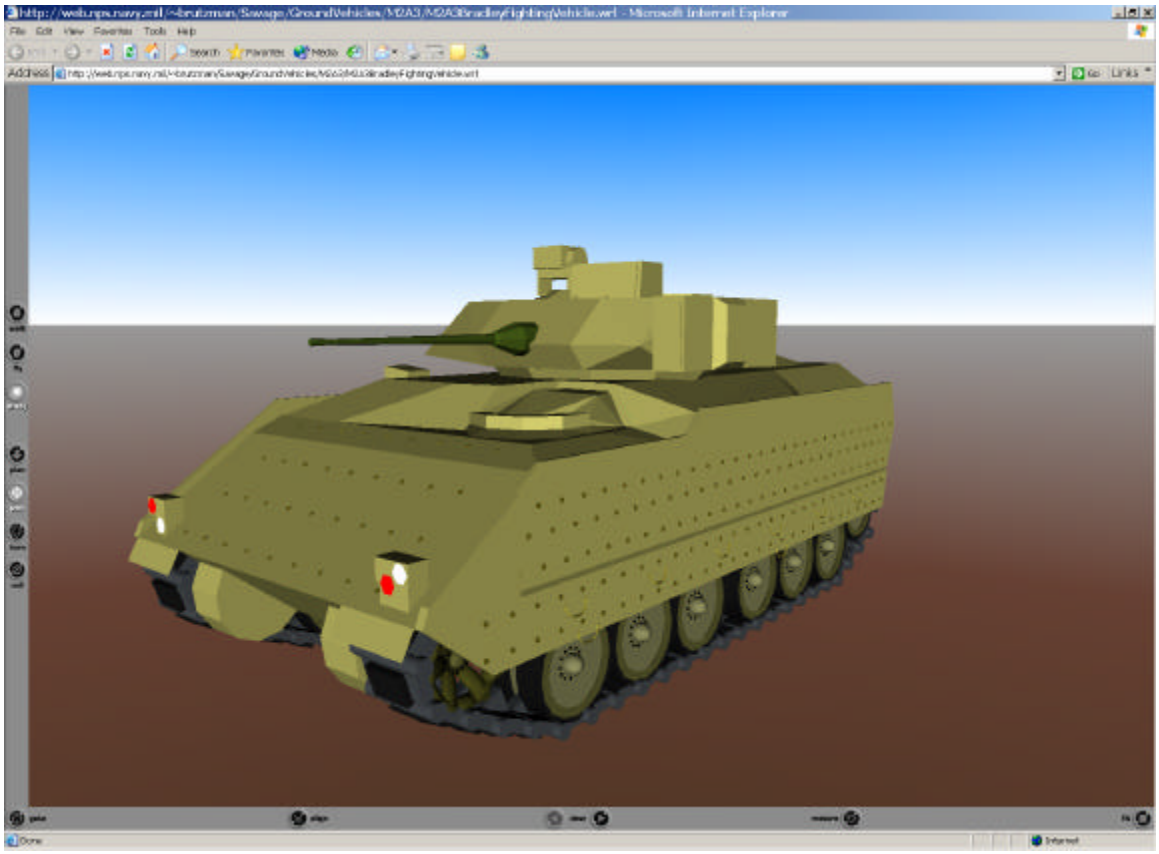

Figure 7. US Bradley fighting vehicle by Renee Burgess [http://web.nps.navy.mil/~brutzman/Savage/GroundVehicles/M2A3/_pages/page04.html]

There are also models of Soviet built equipment. This T-72 tank could be used for enemy forces in a 3D battle.

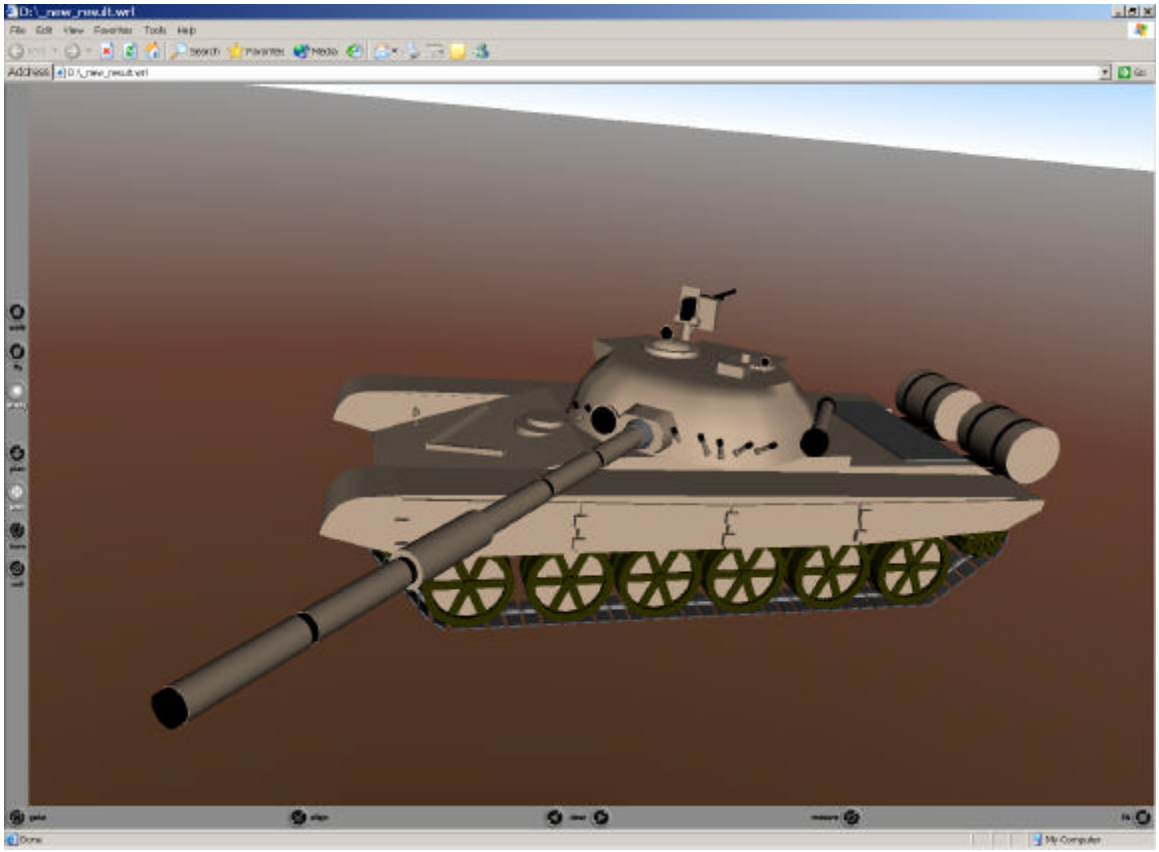

Figure 8. Soviet built T-72M tank by Joseph Chacon [http://web.nps.navy.mil/~brutzman/Savage/GroundVehicles/T72M/chapter.html]

Of course, there are limitations to VRML. This next section addresses four significant limitations to VRML scene graphs that were encountered during the development of this thesis that are not typically addressed in VRML text books.

The first significant limitation is VRML's lack of function or method calls. VRML has script files that can receive data from the scene graph, perform computations, and output data back to the scene graph. In fact, the script files can actually modify the scene graph. However, modifying the scene graph is complicated and still does not allow scripts to communicate with each other like function calls do. Data is passed around between objects in a VRML scene graph through the ROUTE command. Unfortunately, ROUTE commands are one way and anonymous. This means that a point of input for a ROUTE command cannot tell where the call came from and cannot return any data. Function calls would allow several objects in the code to call a function and send data

with each function call. In this thesis, objects are placed on terrain with the object's elevation and orientation determined by code. Typically, there will be several segments of terrain in the scene. Each segment is an independent object. So, an object moving around within the scene can cross over several terrain segments. However, the object's position must be routed to/from a single terrain segment. When the object moves from one object to another, the ROUTE statements involved would have to be changed to point to the new terrain segment. This thesis overcomes this limitation by placing such functionality in a Java script. Thus, how the code in this thesis works is not readily apparent to X3D/VRML programmers by looking at the scene graph. Fortunately, the solution does allow the user to work solely with the VRML code without requiring any further Java programming. Another important caveat for future work is that such code follows VRML 97 scripting conventions.

The second limitation is the limitations imposed on the user because VRML rendering engines run inside a web browser. Web browsers are designed to work with the internet extensively. Since the internet is known to have plenty of viruses, Trojan horses, and other nasty surprises, web browsers have a lot of built in security. One of the primary security measures is to limit the activities that the browser, and thus web sites, can do. For example, web browsers do not allow programs that they are running to access the hard drive directly. Doing so would allow web sites to place viruses on the user's computer at will without the user knowing. This does not mean that there is no access to local disk at all, though. A VRML scene graph can have an Inline command in it that instructs the web browser to load another .wrl file into the current scene. The .wrl file could be on the local hard drive or on the internet and the browser will attempt to complete the task. However, this is indirect access to the hard drive. The VRML programmer can only specify that additional scene graph data needs to be added to the current scene graph with the Inline command. The fact that this sometimes requires the browser to access the local hard drive protects the end user by only allowing the browser, not the VRML programmer, to access the local drive. Likewise, programs running inside a web browser cannot open up sockets on a network to communicate with other computers except under very controlled conditions. The only exception that the author has seen to this policy is with Microsoft's Internet Explorer which allows a program

running under a browser to open sockets when the code for the program is locally stored in a Java jar file that is located on the current PATH environment variable. Normally, though, the program is only able to communicate small messages with the web site the program came from. The result of this is limiting the ability of any program running in a web browser, to include VRML and X3D programs, to request, retrieve, and send data. Since rendering terrain requires a lot of data, this situation hurts this thesis project. The examples included with this thesis place all the necessary code and data on the client computer ahead of time so that the programs run without the user having to reduce the security settings on his or her computer. A new rendering engine called Xj3D, located at www.web3d.org/TaskGroups/source/xj3d.html, can operate as a stand-alone application that can render X3D/VRML, access the local hard drive, and open sockets over the internet. The program is currently still in development at Milestone 8, but it has demonstrated a powerful rendering engine that is fully compatible with existing VRML 2.0 models and scenes. This locally loaded and launched application is able to access both the hard drive and the internet so that terrain data can be downloaded, stored, and used at runtime.

The third limitation is a performance limitation. This issue is definitely arguable, but this author believes that VRML has some flaws that undermine the system's potential performance. Most 3D scenes cannot be built out of primitive shapes such as spheres, cones, cylinders, and boxes. The real world is just simply not set up that way. So, users have to build 3D models of the objects in the world manually. Building these objects requires defining polygons or faces that are used to build whole objects. Most rendering engines give the user options when building these faces. For example, in OpenGL, the user can build objects out of triangle strips, triangle fans, quads, quad strips, polygons, and more. In VRML, the only construct is the indexed face set in which the user defines individual polygons. How to build an indexed face set is covered in more detail in the next chapter. OpenGL gives the user several options because some of the methods of building polygons, such as triangle strips and triangle fans, are rendered much more efficiently than plain polygons. VRML does not let the user help the rendering engine by providing it some optimized polygon constructs. The VRML indexed face set does not even force the user to define polygons that are coplanar and convex. If the user builds

polygons that are not coplanar and not convex, then the VRML rendering engine is forced to break the polygons up into smaller polygons that are coplanar and convex. These features can be helpful to novice graphics programmers, but intermediate to advanced programmers will be much more interested in performance and will be willing to build objects out of more efficient triangle fans and strips if it means better performance. VRML does not even present this as an option.

The fourth limitation is the lack of support for double-precision variables. VRML only supports single-precision variables or float variables. VRML was built to work over the internet which is very slow at transferring data when compared to hard drive and direct memory access speeds. The internet was even slower during the 1990's when VRML was developed. Single-precision variables take only half as much memory as double-precision variables. So, by limiting data to single precision variables, VRML files are smaller and easier to download. This may be part of the reason why VRML only supports single-precision variables. Another possibility is that the designers envisioned VRML being used for relatively small scenes where single-precision values are precise enough to accurately place and render the objects. Whatever the reason for the exclusion of double-precision values, the limitation exists. The result is that defining scenes on a planetary scale cannot be done accurately enough to prevent visual artifacts. Once again, X3D is addressing this limitation. The X3D specification calls for support of both single and double-precision variables. So, once native X3D rendering engines are available, double precision variables will be available. Until then, though, the geoVRML extension to VRML will have to suffice.

2. GeoVRML Extensions

GeoVRML is an extension to VRML developed primarily by Dr. Martin Reddy of SRI. The purpose of GeoVRML is to allow VRML to render scenes on a global scale. The GeoVRML extension can be downloaded and installed royalty free from http://www.geovrml.org/1.1/download. Here are some examples of the types of scenes that can be created using the GeoVRML extension.

Figure 9 shows Squaw Valley built from a height field with texture maps and was taken from http://www.ai.sri.com/~reddy/geovrml/examples/squaw/squaw.wrl. Texture maps alone can make a realistic looking scene. However, placing the texture maps over a

3D representation of the terrain as shown makes the scene much more realistic as the mountains and valleys are shown at their true scale.

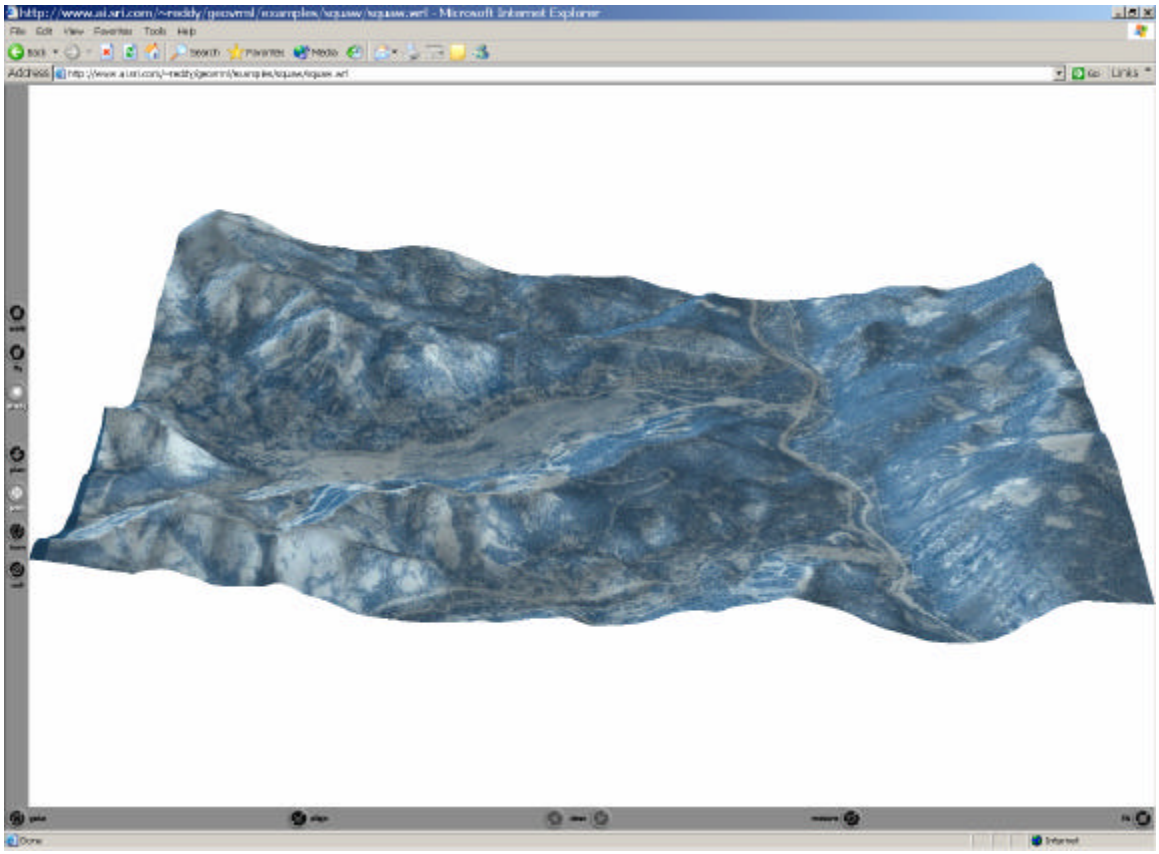

Figure 9. Squaw Valley by Martin Reddy
[http://www.ai.sri.com/~reddy/geovrml/examples/squaw/squaw.wrl]

Rendering global scenes requires double-precision variables, so GeoVRML stores double-precision values as strings and converts them to double-precision variables inside Java code when needed. The solution works quite well, but several VRML had to be replaced by new GeoVRML nodes. These nodes work with the double-precision values stored as strings and the GeoVRML package to build scenes on a global scale. These scenes are then transformed into single-precision values that the VRML rendering engine can display. Basically, the GeoVRML package takes geographic positions specified in latitude and longitude or in UTM, converts them into Geocentric Coordinates, and then translates them into a local frame that can be stored in single-precision variables. This deserves a more extensive look.

First, there are some settings that GeoVRML depends upon to do conversions. The first is the GeoSystem setting. This setting refers to the coordinate system a particular string is in. For example, a latitude and longitude grid uses a GeoSystem of "GD" or "GDC". The GeoSystem can also specify the geoid such as "WE" which is the default geoid and the most common. The final GeoSystem variable would look like "GD WE" or "GD" "WE". Alternatively, UTM coordinates could be specified with "UTM Z10" to specify a UTM coordinate in zone 10 or "UTM" "Z10". Of course, any of the 60 zones can be specified. Finally, a geocentric coordinate can be specified with a GeoSystem of "GCC" or "GC", but this should be avoided because the system is only present to allow the computer to render the scene. GCC coordinates are very difficult for people to work with. Users should always let GeoVRML convert coordinates to GCC so that they are consistent with all other GCC coordinates that GeoVRML automatically generates behind the scenes. Here are some example coordinates with the appropriate GeoSystem.

Latitude and Longitude Example:

geoSystem "GD", "WE"

position "29.7118644 52.6271186 0"

Universal Transverse Mercator Example:

geoSystem "UTM", "Z13"

position "4039260 455220 0"

Notice that each position has three numbers. In the first example, these numbers are the latitude, longitude, and elevation. In the second example, the numbers are the northing, easting, and elevation. I could not find the limit on how many characters can be placed in one string, it could vary from browser to browser, but rest assured that strings can hold a significant number of digits allowing enough precision in the coordinates. Note that all of the numbers in the examples are within the range of values a single-precision variable can hold. In fact, the UTM example uses integers which can be held exactly in integer variables. The catch is that floating point numbers such as single-precision numbers are not stored as integers. Instead, the above UTM coordinates are

stored as 4.039260E6, 4.55220E5, 0.0E0 when stored as float variables, just like the latitude and longitude values. So, the question is how accurate is the fractional value? Here is where the numbers get confusing. The problem is that the computer stores values in binary while people use decimal values. The two systems do not map well when decimal points are involved. For example, if one bit is used to hold the value after a decimal, then it can only hold two values: .0 and .5. If two bits are used, then four values are possible: .0, .25, .5, and .75. Notice that there are gaps between these numbers that can be represented by decimal values. This is where accuracy is lost. Take the example with two bits of precision trying to represent the decimal number .3. The closest representation is .25 which is far enough off to make a noticeable difference. Single-precision numbers use 32 bits to store the exponent and the fraction together. This results in a lot more precision than the 2 bit example, in fact, there is enough precision to represent anything in the screen space of even high end displays of 1600 by 1200 pixels. However, when coordinates are specified on a global scale, single-precision numbers are not sufficient. The bottom line is that with single-precision values, coordinates can only be specified to about 8 meters of resolution when specifying coordinates in terms of latitude and longitude or UTM. This means that when an object's location is specified, it can be displayed up to 8 meters from the exact location specified due to lost precision. What is even worse is that the object's location can move around as the viewpoint changes within the 8 meter radius of its actual specified location. Double-precision variables, however, increase the accuracy almost to the microscopic level of precision at a global level.

The second setting that GeoVRML depends upon for properly rendering terrain on a geographic scale in called the GeoOrigin. The GeoOrigin is a reference point that all the coordinates calculated are offset against. What this means is that this point is converted to GCC space. Then, the X, Y, and Z values of this GCC coordinate are subtracted from the X, Y, and Z values of every point that is created for the scene. This effectively strips away most of the value leaving a fractional value that can be held within a single-precision variable. For example, assume that a GeoOrigin's X value was calculated to be 6,132,248.1643872 and is held in a double-precision variable. A coordinate is then calculated with an X value of 6,132,074.732843. The X value for this

coordinate cannot be held in a single-precision variable without losing precision. However, when the GeoOrigin's X value is subtracted from it, the result is -173.4315442 which can be stored in a single-precision variable with a lot more accuracy. This final value is what is passed back to the VRML rendering engine as the X value for that coordinate. GeoVRML does this translation to every coordinate value including the coordinate values that define where the viewpoint into the scene is. The result is that the particular portion of the earth that the scene is rendering is shifted much closer to the origin of the screen space. This eliminates a lot of the digits in the values leaving them small enough to be stored in single-precision variables. As long as the GeoOrigin chosen is close enough to the coordinates in the scene, the final translated coordinate values will be accurate enough to produce a correctly rendered scene. So, the best choice for a GeoOrigin is one that is close to the center of the scene. However, choosing a GeoOrigin that is located at an extreme corner of the scene is typically close enough. The only remaining question with the GeoOrigin is what to use when the whole planet needs to be displayed? The answer is to use a GCC coordinate with zeros for the X, Y, and Z values. This is the center of the earth and one of the few times when entering a value in GCC coordinates makes sense. This will result in the GeoOrigin's X, Y, and Z values being 0. So, when coordinates have these values subtracted from them there is no change. This results in a loss of precision when casting the double-precision values to single-precision. However, the loss ends up being less than a pixel in screen space when the viewpoint is far enough away from the earth for the entire planet to be displayed at one time.

Now for some details about how the GeoVRML code works. There are script files for most of the nodes in the GeoVRML extension such as GeoLocation, GeoPositionInterpolator, GeoElevationGrid, etc. However, these script files are all dependent upon the geovrml.class Java file to perform their operations and work together. Therefore, this thesis is focused on this particular class file. There are five groups of methods that constitute most of the functionality of the class. First, there are the methods that determine the GeoOrigin. These are the setOrigin and getOrigin methods. The setOrigin method takes a string representing the coordinates of the GeoOrigin and another string representing the GeoSystem. A GCC coordinate is calculated from these values and stored in a class variable. The getOrigin method returns the contents of the

origin as a GCC coordinate. Whenever a GeoVRML object is created, setOrigin must be called to initialize the object before any other operations are used.

The second group of methods is the getCoord and getCoords methods. These methods are used to transform coordinates that are in latitude and longitude or in UTM to GCC. Once the conversion is completed, the current GeoOrigin is subtracted from the values of the GCC coordinate. These methods are the only methods that should be used to determine GCC coordinates. There are three versions of both of these methods that allow the user to specify the parameters in different forms.

Third, are the geoCoords methods. These methods do the opposite of the getCoord methods. These methods take GCC coordinates and transform them back into UTM or latitude and longitude. Once again, any time a GCC coordinate needs to be converted, these methods should be used because they properly reapply the GeoOrigin to the coordinates before transforming them. Also like the getCoord methods, there are three geoCoords methods that accept various parameters.

The fourth group of methods is the getLocalOrientation methods. These methods are not used very often. In fact, the GeoLocation and GeoViewpoint nodes appear to be the only nodes that use these methods. However, the methods are extremely important. These methods calculate a rotation vector and angle that will orient the viewpoint so that the terrain appears right-side-up. The problem is that when a viewpoint is focused on a small area of the planet, the positive Y direction will probably not represent "up". Take for example the South Pole. If you look directly at a globe, the South Pole is at the bottom of the globe and "up" from the South Pole is actually toward the floor of the room the globe is in. Therefore, the viewpoint must be turned upside down when viewing terrain at the South Pole. Likewise, "up" along the equator is parallel to the floor of the same room. Using one of the getLocalOrientation methods returns a VRML rotation node that will orient anything it is applied to so that the object's previous "up" direction (the positive Y direction for most models) will not be pointing in an unrealistic direction.

The fifth and final group of important methods is the VRMLToString methods. These methods convert VRML strings to Java strings. Actually, only the VRMLToString method that takes an MFString parameter is important. For SFString variables, just

simply calling the getValue method of the SFString is sufficient. With the MFString variables, though, an error can result using certain browser plug-ins. Just be careful not to use the toString methods of VRML string objects. These always contain quotation marks which must then be manually stripped out. Using getValue or VRMLToString is much easier.

Working with GeoVRML can be very simple. Simply remember to set the GeoOrigin first. Then use getCoord or getCoords to convert coordinates to GCC so that they can be displayed by the VRML rendering engine, and to always take GCC coordinates back to UTM or latitude and longitude using the geoCoords method. Finally, remember to use the getLocalOrientation method to get a rotation node for viewpoints and objects so that they appear right-side-up in the scene. This is a fairly complete toolkit. All that is really lacking is support for transforming coordinates from UTM to latitude and longitude and vice versa. This thesis adds functionality to the GeoVRML extension to VRML, but all of the additional functionality is at a level above this class. The script nodes developed for this thesis are as dependent upon the GeoVRML class as the nodes provided in GeoVRML 1.1.

3. X3D Specification

X3D represents the next generation of interactive 3D graphics designed for the Web. The specification is compliant with XML which brings benefits such as the ability to translate X3D files to other formats with XSLT. Currently, this is used to translate X3D files to VRML files for display in VRML-enabled web browsers. Likewise, X3D files can be validated using XML schemas. The specification is available at http://www.web3d.org by following the link to specifications. The specification currently defines over 130 nodes for building 3D scenes. Readers are encouraged to visit the site and read the specification. The actual X3D editor is available from the same website at www.web3d.org/TaskGroups/x3d/translation/README.X3D-Edit.html.

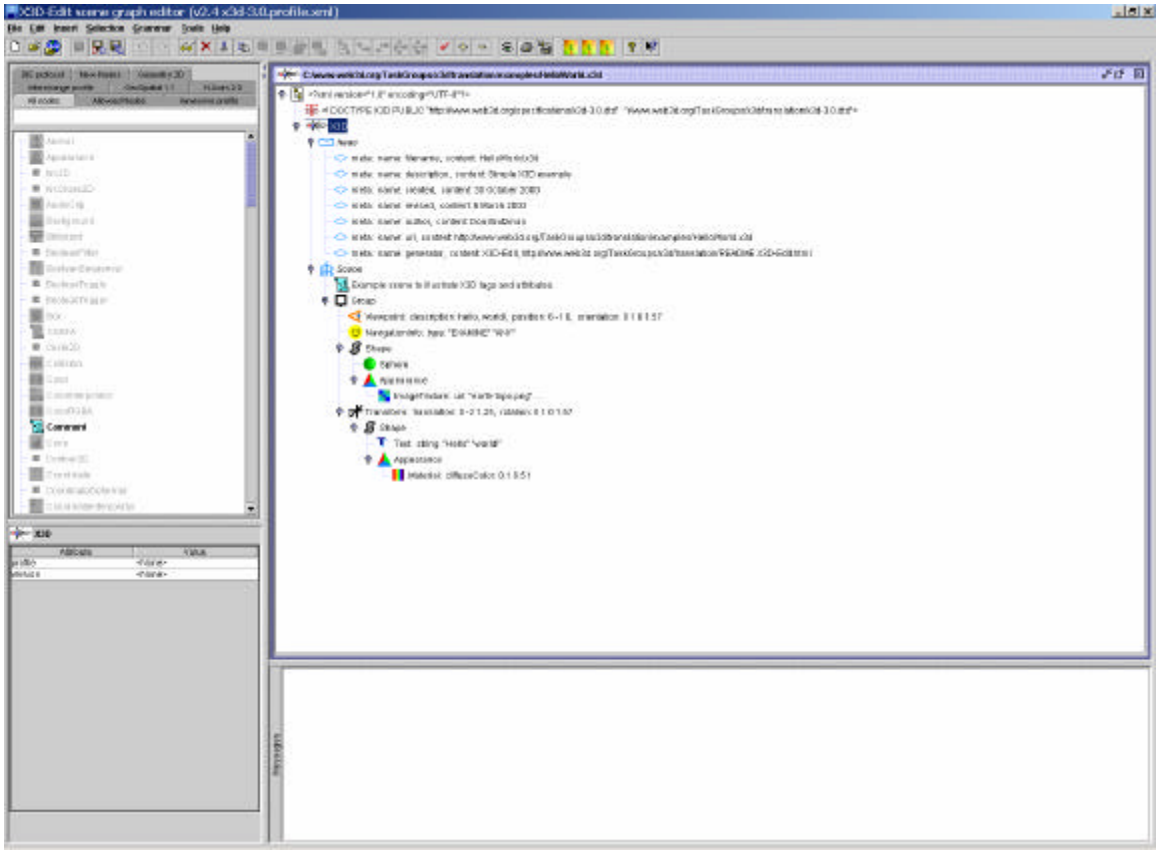

Figure 10. X3D-Edit main screen [http://www.web3d.org/x3d.html] .

A native X3D rendering engine is being built called Xj3D. The project is an open source project whose home is currently located at http://www.web3d.org/TaskGroups/source/xj3d.html. At the time of this writing, version 7 was available with version 8 in a beta state. The project is a native X3D engine.

F. SUMMARY

This chapter discussed how positions on the earth are described using three separate systems. The first two are common geographic referencing systems used all around the world while the third was a computer specific system needed to help the computer render geographic scenes. Next, DTED, a common source of terrain elevation data, was explored as a basis for building 3D models of terrain. XML was then introduced as a technology that can help make accessing and validating terrain data easier

and more universal. Finally, X3D and VRML were discussed. These two rendering engines are needed to actually create the visual displays from the 3D models that this thesis deals with.

III. TERRAIN RENDERING ALGORITHM IMPLEMENTATION

A. INTRODUCTION

This section details the code that was developed for this thesis. First, the GeoManager is introduced which allows multiple pieces of terrain to be loaded and used with minimal effort and which allows objects to access the data in the pieces of terrain through a GeoLocation3 object. Second, the GeoTerrainGrid is described which does the majority of the work covered in this thesis in particular, building the 3D model of a section of terrain and calculating the elevation and orientation of arbitrary positions on this terrain. Finally, the GeoLocation3 node is covered which allows objects to use the technology developed.

B. GEOMANAGER

In the previous chapter, a Java solution to VRML's lack of support for function calls was introduced. This solution is the GeoManager object. The GeoManager object provides a way for the GeoLocation3 object to locate a GeoTerrainGrid that holds terrain data for a specific location defined in GDC or UTM coordinates. For this to work, the GeoManager must be able to reach all the GeoTerrainGrids in the scene and every GeoLocation3 must be able to reach the GeoManager. The system also depends upon only one GeoManager existing within the scene. In order to ensure that only one GeoManager exists, the constructor had to be private. With a private constructor, instances of GeoManager can only be created from within the GeoManager class. Thus, a public, static class is needed that creates one and only one instance of a GeoManager. The getGeoManager method does this. The first object that calls getGeoManager causes a new GeoManager to be created and returned. All subsequent calls to getGeoManager return the same GeoManager instance without creating a new one. Every GeoTerrainGrid and GeoLocation3 object can call getGeoManager because the method is static.

Inside the GeoManager class, there are methods written specifically for GeoTerrainGrids and GeoLocation3 objects. For GeoTerrainGrids, there is addGrid which allows a GeoTerrainGrid to announce its existence to the GeoManager. Every time this method is called, the GeoManager stores a reference to that GeoTerrainGrid in a

vector. Every GeoTerrainGrid must call this method when created so that the GeoManager can reach every GeoTerrainGrid in the scene. The method for GeoLocation3 objects is the getGrid method. This method takes a location as a Gdc_Coord_3d, which holds latitude and longitude values, and returns a GeoTerrainGrid that contains that location. This allows GeoLocation3 to get a GeoTerrainGrid and use it to determine the proper elevation and orientation. These function calls are shown in Figure 11. There are more methods in GeoManager, but those were specifically written for an experiment where a military simulation was listened to and displayed in 3D. The experiment was successful for a few entities, but more work is needed on dynamically loading and unloading large numbers of GeoTerrainGrids and entities such as tanks and helicopters before the code will be useful to the military. The SAVAGE website will contain future versions of this code to address these issues.

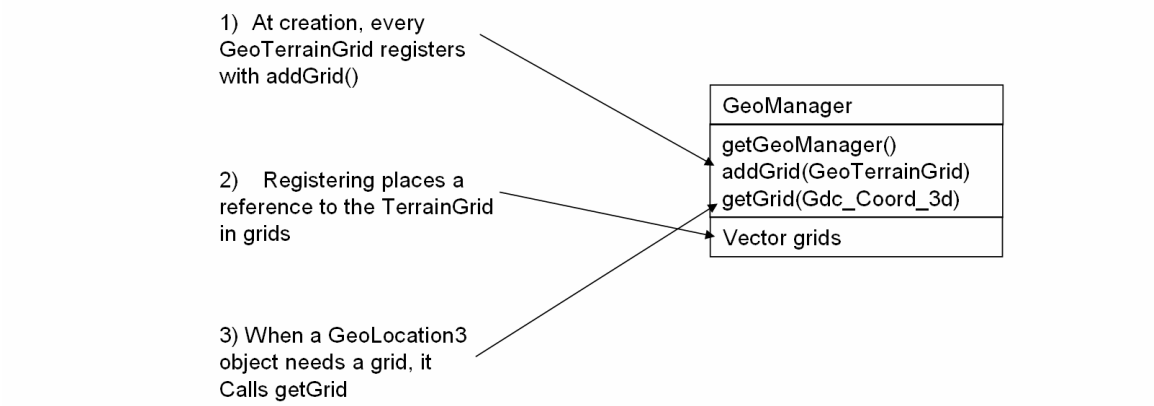

Figure 11. Depiction of how GeoManager is accessed

C. GEOTERRAINGRID NODE

The GeoTerrainGrid object is responsible for actually rendering a piece of terrain based on a height grid and for providing elevation and orientation data for objects that reside within the boundaries of the grid. There are three major functions that this code performs. The first is creating an indexed face set that represents the terrain described by the height field. The second is determining the elevation at an arbitrary point within the boundaries of the grid. The third and last is determining the proper rotation to apply to an object so that it is placed properly on the terrain. Each of these functions is covered in its own section. The GeoTerrainGrid is a direct descendent of the GeoElevationGrid

contained in the GeoVRML extension to VRML. All of the original GeoElevationGrid code is still in GeoTerrainGrid, although some of it had to be commented out with new code replacing it. All of the changes are clearly marked. Here are the major changes made in GeoTerrainGrid. First, GeoTerrainGrids are capable of working with GeoManagers so that objects can travel across several GeoTerrainGrids effortlessly. Second, the elevation of any location within the boundaries of the GeoTerrainGrid can be estimated and retrieved (if the location coincides with an elevation posting, then the exact location is given). Third, a rotation vector can be retrieved that will rotate an object to align its up vector with the normal of the terrain at that location. Finally, the indexed face set, coordinate list, and height fields are not stored at class level anymore. Instead, they are retrieved when needed. This significantly reduces the memory footprint of a GeoTerrainGrid over a GeoElevationGrid, but does incur some overhead every time data needs to be retrieved. These savings quickly become significant in the example GeoTerrainGrids built in this thesis. Every sample GeoTerrainGrid in this thesis covers an area that is approximately 1.8 km by 1.8 km in DTED level 2 data. The height fields have 3,721 entries total including both axis. These are single-precision floating point values taking 4 bytes each. Thus, each height field fills 14.5 kilobytes of memory. The indexed face sets built from these height fields have 3,721 coordinates, one coordinate built from each height field value. Every coordinate has an X, Y, and Z value that is a 4 byte, single-precision floating point value. Thus, the coordinate list occupies 43.6 kilobytes of memory. The coordinate list of every indexed face set requires 4 integers for each triangle. Since the height array builds a grid that is 60 squares by 60 squares, 7,200 triangles are needed. This is another 112.5 kilobytes of data. Thus, each GeoTerrainGrid built for this thesis requires 14.5 kilobytes for the height array, 43.6 kilobytes for the coordinate list, and 112.5 kilobytes for the coordinate list. The total is 170.6 kilobytes for a 1.8 km by 1.8 km area. When the GeoTerrainGrid is built, there are two copies of all these data structures, one in the VRML data and one in the Java code. To save memory, the Java script code drops its copy of the data when finished saving 170.6 kilobytes. The multiple grid example has 16 GeoTerrainGrids and saves 2.73 megabytes of memory using this technique. Of course, one copy of the data still exists within the VRML code and is retrieved by the Java script code when needed.

Here are two screen shots of GeoTerrainGrids. Figure 12 shows shaded terrain.

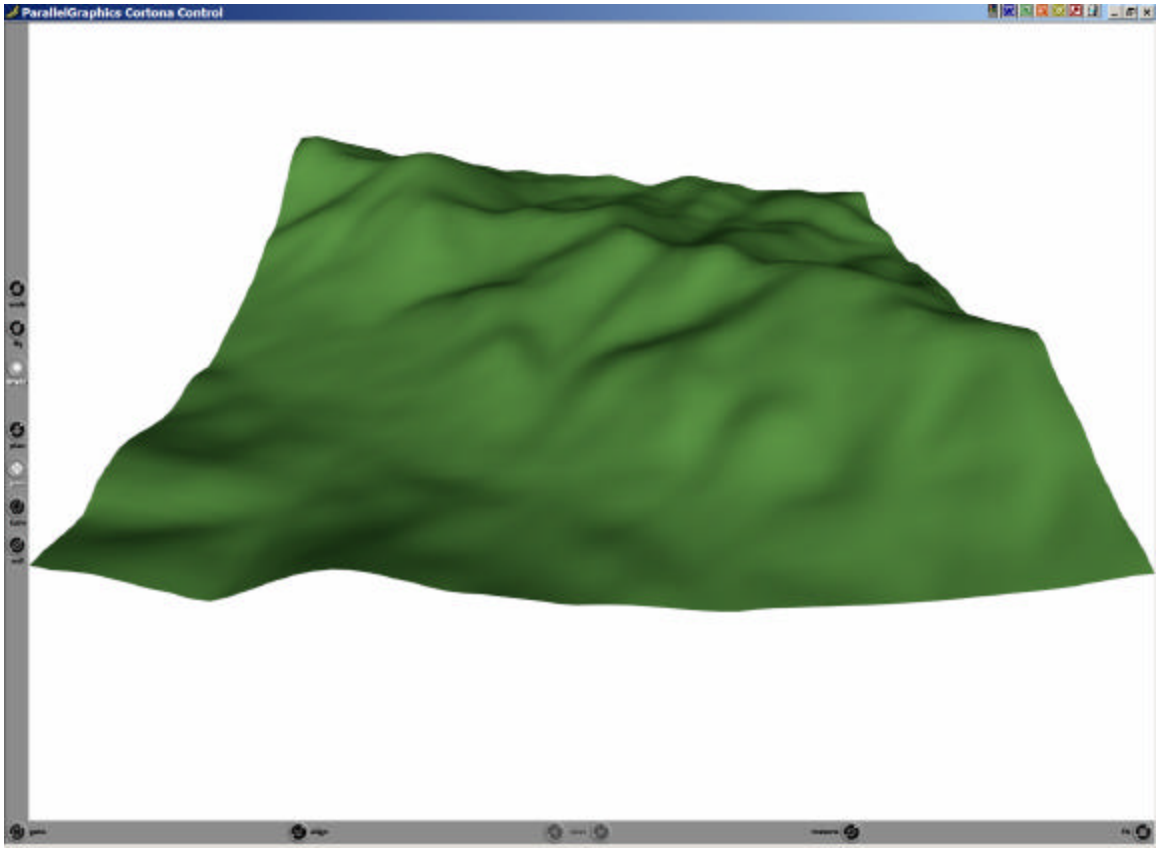

Figure 12. GeoTerrainGrid example with shaded terrain

Figure 13 shows the same terrain in wire frame mode. In this view, how the elevation grid is built out of triangles is apparent. Notice how the shaded terrain is very smooth despite being created by triangles with straight edges.

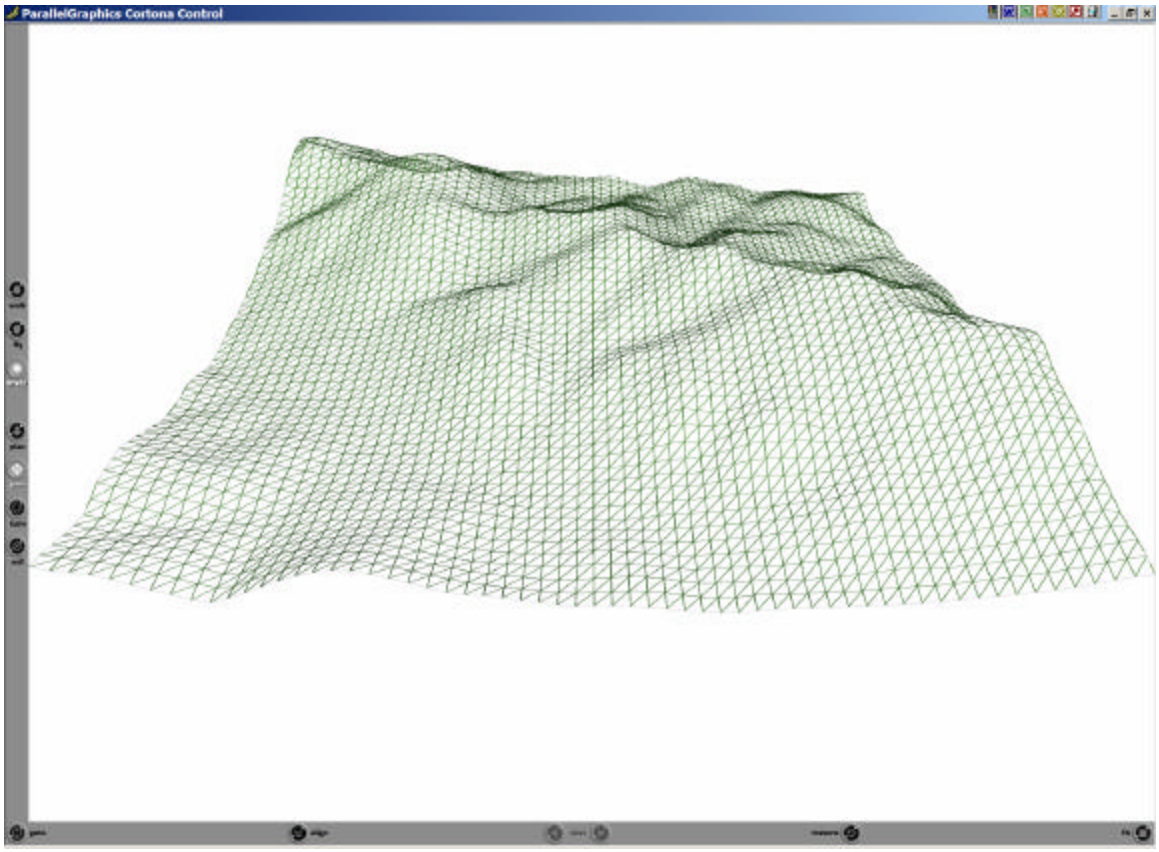

Figure 13. GeoTerrainGrid shown in wire-frame mode

1. Building an Indexed Face Set from a Height Array

A height array is simply an ordered list of numbers where every number corresponds to a measured elevation point on a surveyed piece of terrain. The height values are typically taken at evenly spaced intervals that coincide with one of the geographic coordinate systems described in the previous chapter such as latitude and longitude. Note that GeoCentric Coordinates do not work because there is no elevation component. The three values of X, Y, and Z are all required to determine a position. None of these corresponds to height. The height is implicitly contained in the coordinate, but creating coordinates based only on a height field and a start point is not possible. Therefore, the height arrays in this thesis must be based on latitude and longitude values or UTM values. The GeoElevationGrid that the GeoTerrainGrid descends from has this same requirement. If the height values from the height array are arranged in accordance with their geographic coordinates, then a checkerboard pattern forms with every corner of each square of the checkerboard being a height value. Building an indexed face set from

this height array is simply a matter of connecting the dots of height values in this checkerboard pattern in the proper order.

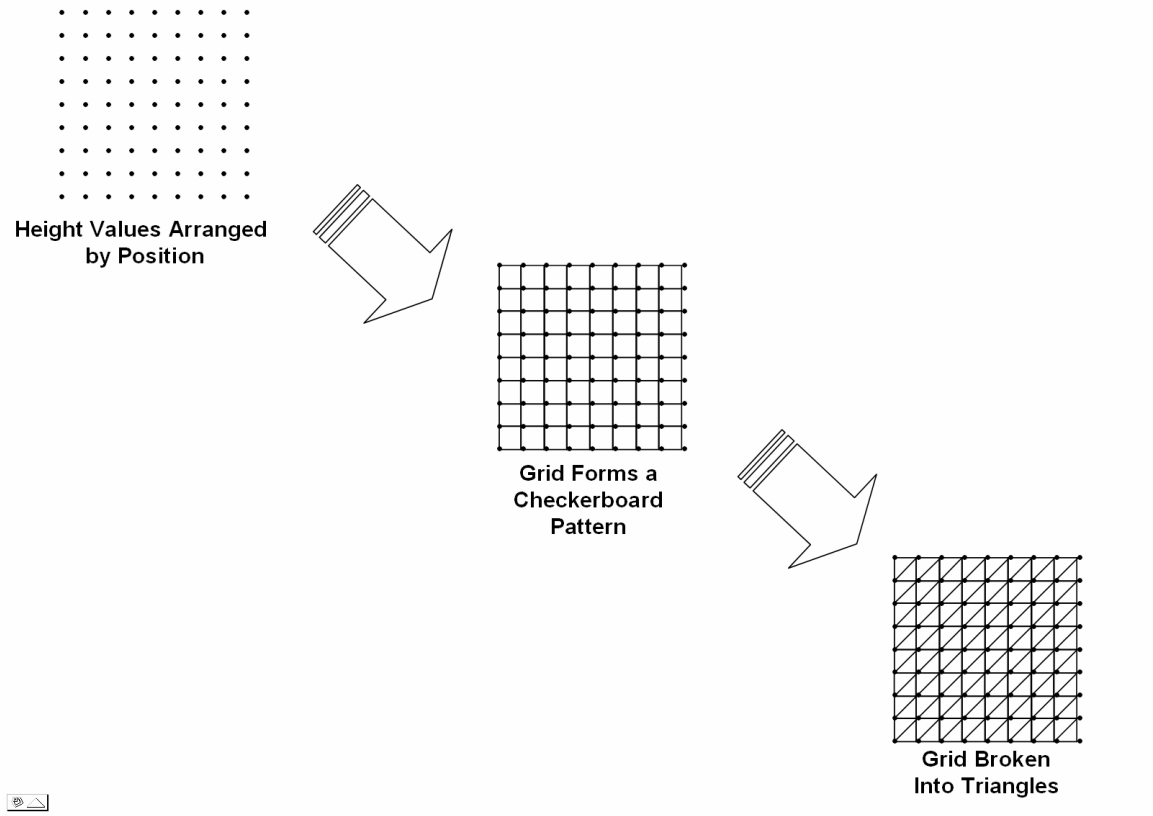

Figure 14. How a height array is turned into an indexed face set

Determining the proper order for connecting the points of an array of height values requires some knowledge of computer graphics. In order for a computer to render an indexed face set, each face in that set must represent a convex polygon where all vertices are coplanar. Some graphics engines, such as VRML discussed in the previous chapter, relax these rules for the end user, but these engines are then forced to break up the polygons provided by the user into smaller polygons that are convex and coplanar. The simplest method for building polygons that are guaranteed to be coplanar and convex is to build triangles. When building an indexed face set from a height array, individual triangles, triangle strips, or triangle fans are generally used. This thesis uses individual triangles because VRML does not explicitly support triangle strips or fans. The individual triangles are all still part of an indexed face set and are connected to each other

because they share vertices. Therefore, the final rendering looks the same as one from triangle strips because the individual triangles were built in the same fashion. Each square of the before mentioned checkerboard is divided into two triangles. The first triangle goes from the bottom-left corner to the upper-right corner to the upper-left corner. The second triangle goes from the bottom-left corner to the bottom-right corner to the upper-right triangle. When every square in the checkerboard is divided up in this fashion, a complete indexed face set is built that when sent through a rendering engine will graphically depict the terrain in 3D space.

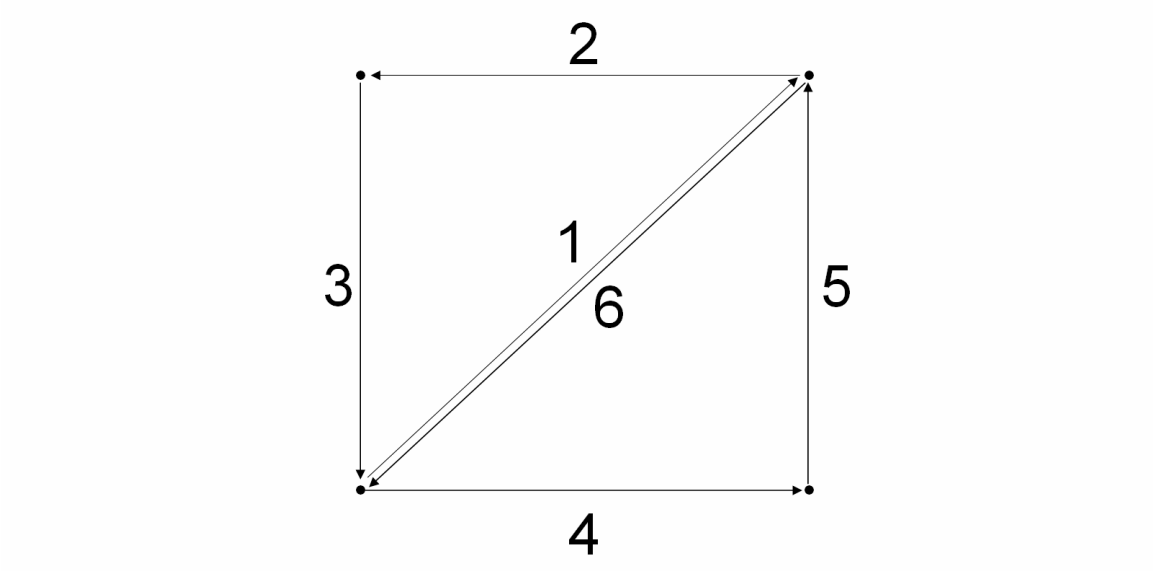

Figure 15. Building a grid square as two triangles

Building the indexed face set is accomplished using two *for* loops with one nested within the other. These *for* loops step through the entire array of height values one value at a time building one coordinate for each elevation posting. The *for* loop uses a helper method in the GeoVRML class called addCoord. The method is simple, it takes the geoGridOriginArray passed to it, which represents the south west corner of the GeoTerrainGrid that must be set by the user when defining the GeoTerrainGrid node, and adds the values for the change in the X and Z directions. All this method does is fill the array passed to it called new_coord. The geoGridOriginArray's X and Z values are added to the values of the offsets for X and Z passed in. The elevation is also the addition of the elevation from the height array and the elevation of the

geoGridOriginArray. So, if DTED is used, then the user must always be sure to set the elevation of the geoGridOriginArray to zero to prevent artificially raising the terrain elevations. These additions are so simple that placing them in a special method in another class seems inefficient when compared to just placing the three lines of code in the method in the nested *for* loop instead. Being in a nested *for* loop causes this method to be called potentially thousands of times. Eliminating the overhead of thousands of method calls makes the code execute faster. After addCoord calculates the correct values for X, Z, and elevation, the getCoord method is used to convert this coordinate to a Geocentric coordinate. This converts the coordinate to GCC and applies the GeoOrigin so that the value can be stored in a single-precision variable without losing significant precision (considering the GeoOrigin is close enough to the coordinate). The coordinate is then stored in an array that holds all the coordinates. Likewise, a list of texture points is also updated. However, the texture points are confusing because specifying such a list is usually only needed if the texture map is not supposed to be mapped exactly to the indexed face set. In GeoElevationGrids, the texture is always mapped exactly unless the user built a custom set of texture coordinates and placed them directly in the GeoElevationGrid or GeoTerrainGrid. Thus, if the user does not define a custom texture coordinate list, then the one that is generated simply does the default behavior for an indexed face set. In fact, when the code was temporarily commented out, nothing changed. However, the author cannot verify that no cases where this code is needed exist. Likewise, to ensure backward compatibility with the GeoElevationGrid, the code was left in.

The next step is building a list of indices for the indexed face set. All of the coordinates have now been translated into GCC and have had the GeoOrigin applied to them to maintain precision. What is left is to specify which of these coordinates to use to build polygons. For example, the first triangle is built using coordinate index 0 which was the first coordinate built. The second vertex is to the right one column of values and up one row of values. Exactly what index value is associated with this vertex depends upon how many X values are in the height array. The third vertex is directly above the first coordinate index and directly to the left of the second coordinate index. The fourth coordinate is a flag value of -1 which tells the VRML engine that this is the end of the

first polygon. The VRML engine automatically closes the polygon by connecting back to the first vertex defined, making a triangle. The fifth coordinate index refers to the first vertex of the second triangle for each square defined by the height array. This is equal to the first coordinate vertex of the first triangle. The sixth coordinate vertex is the coordinate directly to the right of the fifth coordinate vertex. The seventh coordinate vertex is directly above the sixth. Finally, the eighth coordinate vertex is another -1 flag signaling the end of the polygon. Thus, a square defined by four members of the height field has now been broken into two triangles. This is repeated for every such square in the GeoTerrainGrid. Figure 16 shows the order in which the vertices of the grid square are referenced, remember that two -1 entries are also recorded to signify the end of a polygon in VRML. The coordinate list and the coordinate index list define the indexed face set that represents the terrain. Thus, both are passed back to the VRML rendering engine along with the texture coordinate list.

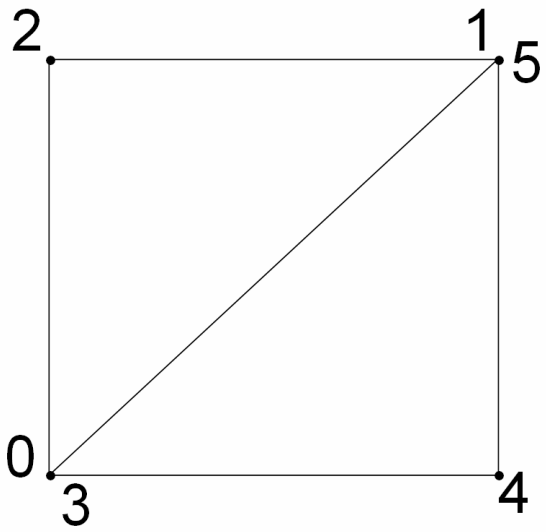

Figure 16. Depiction of building coordinate index list

Before leaving the topic of building the indexed face set, the topic of explicitly defining the rendering of the indexed face set needs to be presented. The original GeoElevationGrid defined the terrain using grid squares. Since these grid squares are rarely coplanar, the underlying rendering engine divided the grid square into two triangles. There are three systems for dividing these grid squares up shown in Figure 17.

Figure 17a shows the grid squares before they are divided, which is how the original GeoElevationGrid defined the terrain. This system cannot be rendered as it stands. The grid squares must be broken into triangles to guarantee that the polygons are coplanar and convex. Figure 17b shows the same grid squares divided into triangles by adding a line segment to each grid square that starts in the lower left corner and ends in the upper right corner of the grid square. This is how the GeoTerrainGrid code builds the terrain representation. With this breakdown, the final rendering has been explicitly defined and cannot be altered by the underlying rendering engine. Figure 17c shows the same technique, but divides the grid squares by connecting the upper-left and lower-right corners. Finally, Figure 17d shows an arbitrary mix of dividing up the grid squares. This is how the terrain of a GeoElevationGrid is rendered after the grid squares are broken up by the rendering engine. This pattern makes it impossible to know exactly how a piece of terrain is being rendered and can cause artifacts.

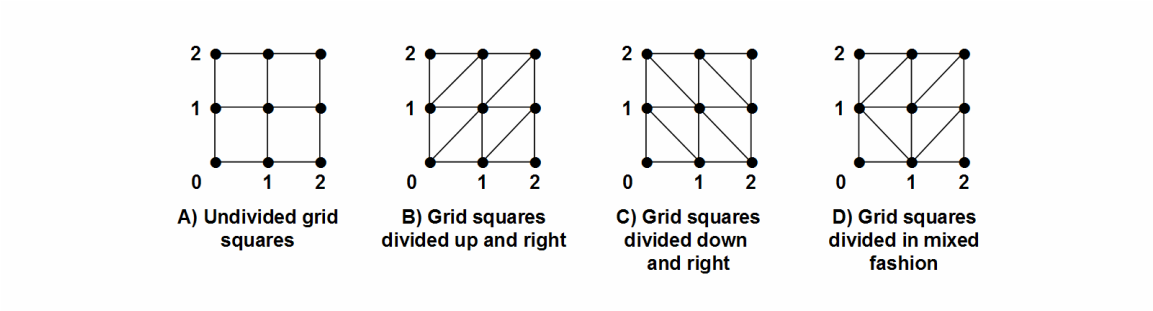

Figure 17. Explicitly defining terrain rendering

Figure 18 depicts a typical problem with not defining the terrain rendering explicitly. The grid square shown has high elevation postings in both its upper-left and lower-right corners while the lower-left and upper-right posting are low elevation postings. The problem is determining if this pattern of elevation postings represents a ridgeline (Figure 18b) or a saddle (Figure 18c). With DTED data alone, the correct rendering cannot be determined, but it can at least be consistent. With the GeoElevationGrid, the rendering could switch between Figure 18b and Figure 18c whenever the viewpoint moved. With the GeoTerrainGrid, Figure 18c is always

rendered. This allows determining the correct elevation and orientation of the rendered terrain at any location.

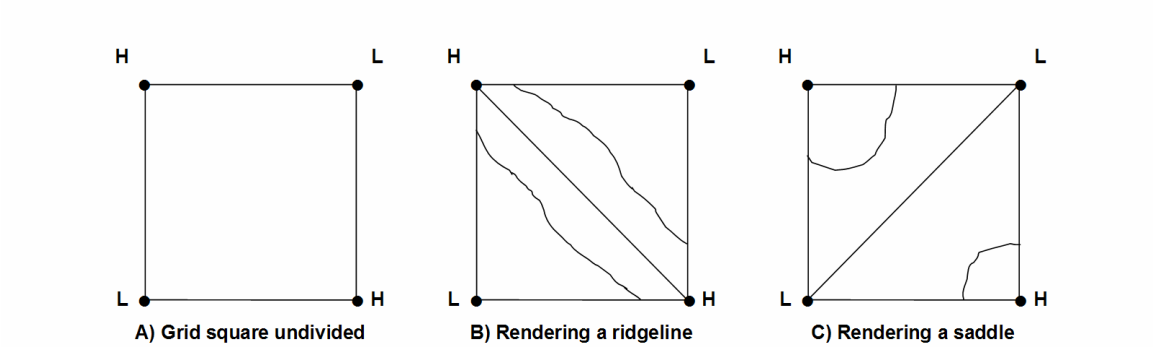

Figure 18. Explicitly rendering terrain

2. Calculating Elevation at an Arbitrary Point

The first step to calculating the elevation at an arbitrary point is to determine which GeoTerrainGrid the point falls within. This task falls to the GeoManager object. The getGrid method accomplishes this by simply going through the list of GeoTerrainGrids and calling the checkBounds method of each one. The GeoTerrainGrids are not in any specific order other than the order in which they were created. Therefore, the search is simply a linear search with an average performance of having to check half of the GeoTerrainGrids before finding the correct one. The worst case scenario is that there is no GeoTerrainGrid that covers the coordinate in question which requires every GeoTerrainGrid to be checked before determining that the coordinate is not covered. Obviously, this could be improved upon, but the linear search was adequate for the scope of this thesis. The checkBounds method of each GeoTerrainGrid only checks latitude and longitude values. So, when getGrid is passed a coordinate, it must be in latitude and longitude. To ensure this, the method only accepts a Gdc_Coord_3d as its argument which is a coordinate in latitude and longitude. Support for other methods could be added. However, UTM grids, the only other coordinate space available in GeoVRML, are easily converted to GDC before being passed to the getGrid method of GeoManager. This will be discussed further under the GeoLocation3 section.

The second step to calculating the elevation at an arbitrary point is to determine which polygon within the GeoTerrainGrid the coordinate falls within. This is handled by

the getElevation method of GeoTerrainGrid. This is a two step process. The first step is to determine which grid square of the height array this point falls within. This is done by determining the values of partialX and partialZ. These variables are integers that represent how far to travel through the height array in the X and Z direction to reach the lower left corner of the grid square that holds this coordinate. The calculation is easy. The value of the south west corner of the GeoTerrainGrid is subtracted from the coordinate. The result is divided by the spacing value between values in the X direction and values in the Z direction. Only the integer portion is kept. Knowing which grid square is not enough, though. Each grid has two triangles and the coordinate can only be located within one of them.

Thus, the third step is determining which triangle within the known grid square holds the coordinate. The variables fractionX and fractionZ are used to do this. First, the variables are set to the fractions that were dropped to get the partialX and partialZ values. These fractions are compared to determine whether the coordinate is in the upper left triangle or the lower right. However, both triangles use the lower left corner of the grid square as their first coordinate. So, the first value of the triCoords array is filled with the coordinate of the lower left grid point. The elevation is looked up in the height array, but the other two coordinates are calculated. It does not matter of the coordinate system is UTM or GDC. In fact, any coordinate system that has elevation as one of its three components will work with this code. The *if* statement determines which triangle to use. If fractionX is greater than fractionZ, then the coordinate falls in the lower right triangle. Therefore, the lower right corner of the grid square is the second coordinate while the upper right coordinate is the third. Otherwise, the upper right corner is the second coordinate and the upper left corner is the third coordinate. Technically, keeping these points in counter-clockwise order is not critical for determining an elevation, but it is always a good practice.

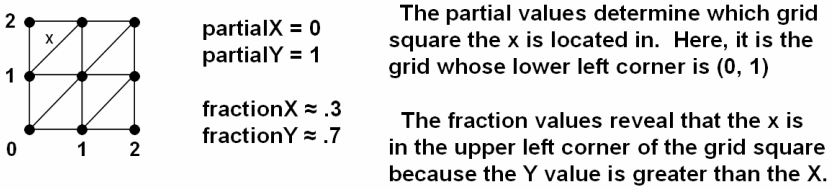

Figure 19. Determining which polygon a point lies within

These three coordinates define the plane that holds the coordinate needed. The normal of this plane is needed to determine the proper elevation of an arbitrary point. First, get two vectors from the three coordinates. This is simple subtraction. The first vector is the first coordinate minus the third coordinate. The second vector is the first coordinate minus the second coordinate. These vectors have an X, Y, and Z component, of course. The cross product of these two vectors is the normal vector for the plane. This normal along with one of the coordinates can be used to determine any other point on the plane. Apply the dot product to the normal and the difference between the known point and the unknown point. This dot product must equal zero. Here is the equation:

$$n \cdot (r - r_0) = 0$$

where n is the normal, r is a known point, and r_0 is the unknown point. Since the only unknown is actually the elevation value of the unknown point, there is only one variable to solve for. So, a simple calculation retrieves the elevation which is then returned to the calling method.

3. Calculating Orientation at an Arbitrary Point

Calculating the orientation at an arbitrary point is similar to calculating the elevation, but GCC coordinates must be used. This sounds strange because GCC coordinates could not be used to determine the elevation because there was no elevation value to solve for. Elevation was embedded within the X, Y, and Z values. Any coordinate system that has one of its three values as elevation will not work for determining the proper elevation. The reason is that these systems are two dimensional systems – even latitude and longitude because it is not 3D until it is converted from spherical coordinates to X, Y, and Z values in 3D space. Think about having elevations that are all positive, i.e., almost any land area of the planet. Higher elevation is always a

greater positive value. But in the southern hemisphere, increasing elevation could actually cause the X, Y, and Z components of the GCC coordinates to become greater negative numbers. So, GCC values are needed which means getting the coordinates for the polygon from the coordinate list of the indexed face set instead of from the height field.

Determining the grid square that the coordinate is in and the triangle within that grid square is the same as it was for finding the elevation in the last section. So, it will not be repeated. However, when retrieving each individual coordinate, the value of the GeoOrigin must be added to that coordinate. Remember that the GeoOrigin was subtracted from each GCC point after it was converted from UTM or latitude and longitude to preserve precision when cast to a single-precision value. This origin must be added back in to get the actual GCC location instead of the translated GCC location. Of course, this means using double-precision variables to hold the GCC location. Calculating the normal is also the same as it was when determining the elevation. First, two vectors are found from the three coordinates, and then the normal is found by taking the cross product of these two vectors. In this case it is critical that the coordinates be in counter-clockwise order and that the vectors be determined from taking the difference of the first coordinate and the second coordinate and then by taking the difference of the first coordinate from the third coordinate. If this formula is not followed, then the normal could be pointing directly into the terrain instead of directly out of the terrain and the rotation value will flip the object upside down on the terrain.

The next step is to take the cross product of the normal vector and the vector that represents ‘up’ for the object. In VRML, ‘up’ is defined to be the positive Y axis. This thesis assumes that all objects follow this standard convention. However, the code could be modified later to use a different value or to allow the user to define an ‘up’ vector. Having the user enter in an ‘up’ value seems to be an unnecessary complication, though, so the VRML standard of the positive Y axis as ‘up’ was used. This cross product gives a vector perpendicular to both the normal of the terrain and the ‘up’ direction of the object. The object can rotate around this axis to bring its ‘up’ vector to coincide with the normal vector of the terrain. All that is left is to determine how far around this vector to rotate. Of course, this rotation vector will have to be cast to single-precision values

which could lose precision. Therefore, the vector is normalized first by dividing each of its components (X, Y, and Z) by the magnitude of the vector. The normalized rotation vector should always work fine even after being cast down to single-precision.

Calculating the angle of rotation to go with the vector is not too difficult. The cosine of this angle is equal to the dot product of the 'up' vector and the normal vector divided by the magnitudes of the two vectors. Mathematically, the formula looks like the following:

$$n \cdot u = ||n||*||u|| * \cos F$$

where n is the normal vector of the terrain, u is the 'up' vector (0, 1, 0) in VRML, and F is the angle between the vectors. Solving for F is simple. See the code for a working example.

D. GEOLOCATION3 NODE

Now that the GeoTerrainGrid is capable of determining the proper elevation and orientation for an arbitrary point, a node to apply these to objects is needed. That is what the GeoLocation3 node does. The reason for the 3 at the end of the name is that there already is a GeoLocation node and a GeoLocation2 node in the GeoVRML package. Therefore, this proposed new node is GeoLocation3. The node functions identically to the original GeoLocation as its default value. However, there are two additional Booleans called autoElevation and autoSurfaceOrientation that the user can set. When autoElevation is true, the coordinates of the GeoLocation will always be adjusted to the surface of the terrain at that location. There are three situations that cause the geoCoords to be set to the terrain. The first is at initialization if autoElevation is true. The second is anytime that the set_geoCoords event is fired and autoElevation is true. Thus, if a route is set up that continually updates the location of a GeoLocation3, then the elevation will be set automatically every time. The third situation is anytime that the set_autoOrientation event is fired and set to true. However, if the event is called again and the value is set to false, then the original elevation will not be restored.

The autoSurfaceOrientation variable determines whether the objects contained within the GeoLocation3 construct are oriented to the terrain or to the local frame. Orienting the object to the terrain makes the object appear to be resting on the ground.

Imagine a car driving up a hill. The car tilts so that the front of the car is higher than the back of the car. To get this behavior from objects, the autoSurfaceElevation Boolean must be set to true. Figure 20 demonstrates this behavior. Setting the variable to false causes the object to be oriented to the local frame only. This is similar to a person walking up the same hill. The person will still be standing straight up.

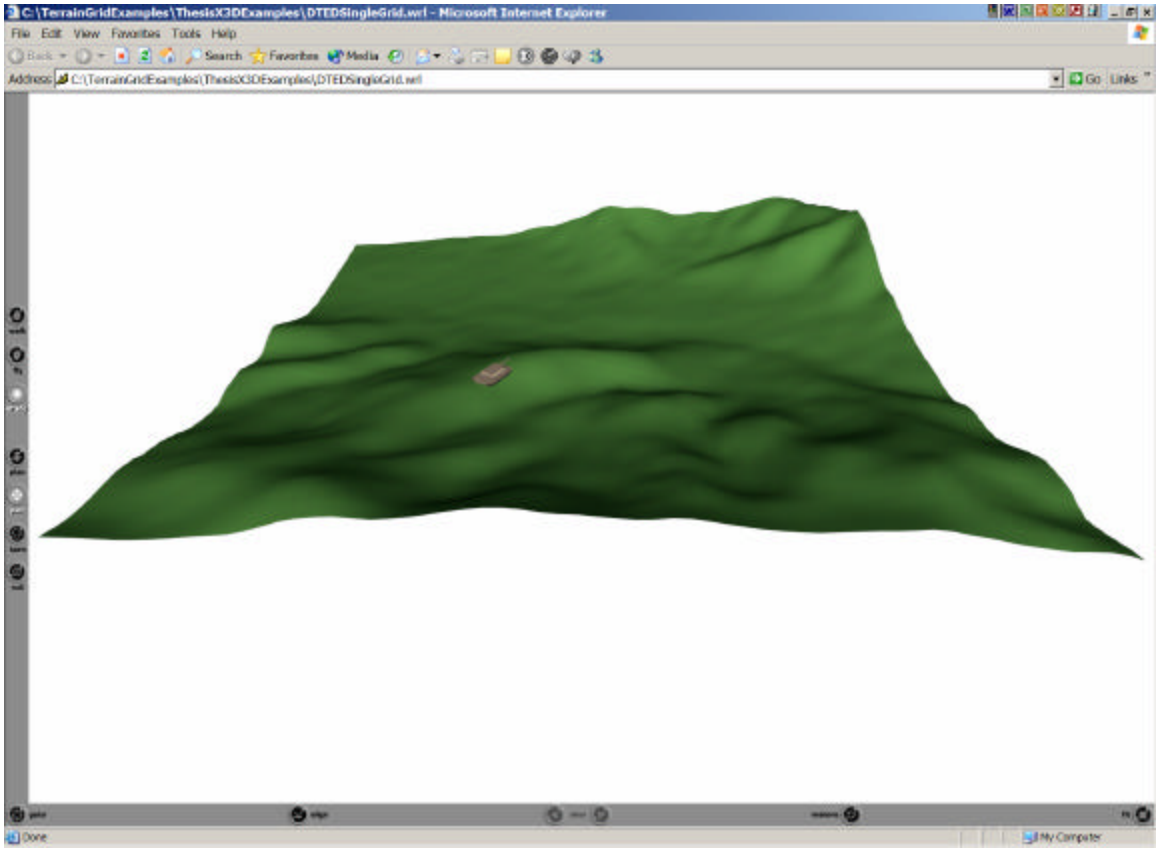

Figure 20. A GeoLocation3 node orienting an object to terrain

The autoElevation and autoSurfaceOrientation variables can both be set to true or false individually to replicate specific behavior. Setting both to true would work well for cars and vehicles that always stay on the ground and tilt with the terrain. Setting both to false would simulate a helicopter that does not stay on the ground and does not tilt based on the terrain beneath it. A building setting would have autoElevation set to true and autoSurfaceOrientation set to false. This way, the building sits on the ground, but stands

directly up toward the sky. The last combination is autoElevation set to false and autoSurfaceOrientation set to true. This combination is strange, but someone may come up with a use for it.

Earlier in this thesis, VRML's lack of function calls was addressed as a limitation because an object can only interact with one GeoTerrainGrid at a time via a route statement. If the object crosses to another GeoTerrainGrid, then the scene graph must be altered. Figure 21 shows that objects using a GeoLocation3 node do not have this limitation because the Java class files handle a function calling mechanism within the Java code automatically. All objects that use a GeoLocation3 node for placement will automatically recognize every GeoTerrainGrid currently loaded into the scene and call upon them for elevation and orientation data when needed.

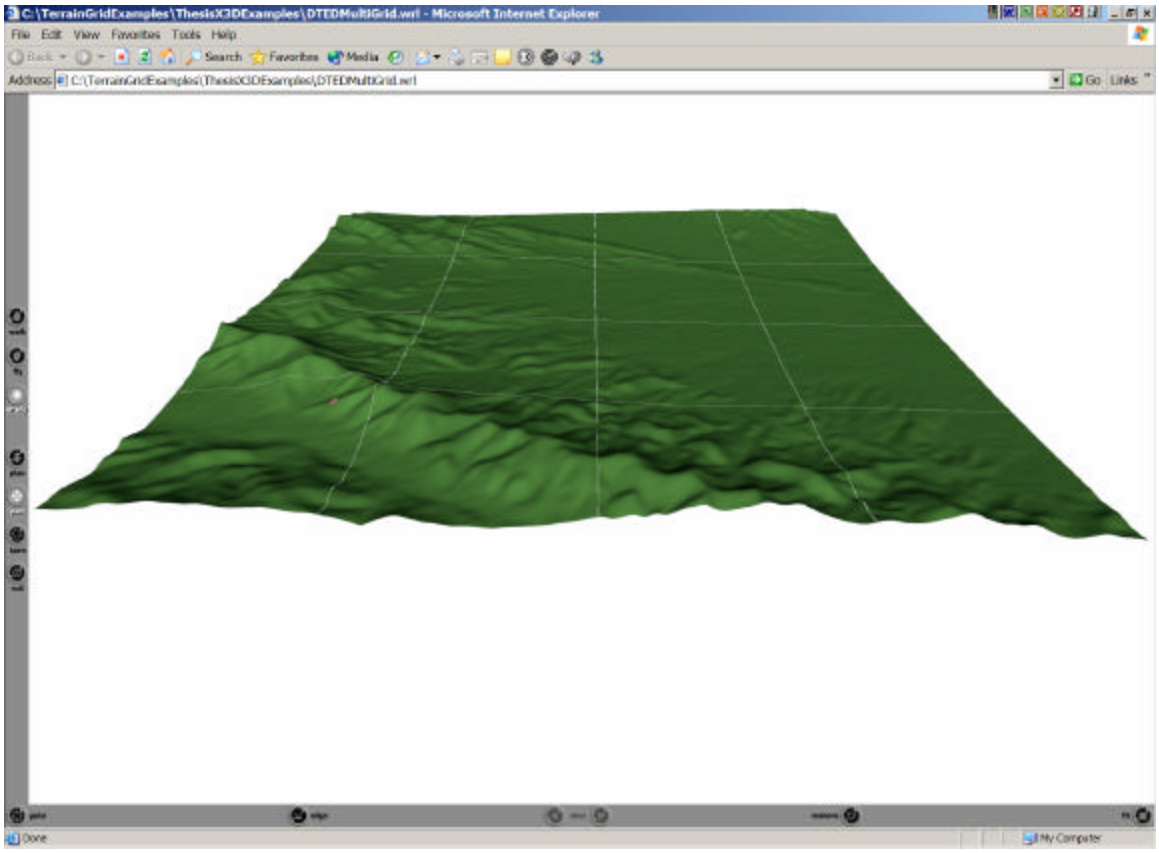

Figure 21. A GeoLocation3 node spanning multiple GeoTerrainGrids

E. SUMMARY

This chapter described the three Java class files that demonstrate the technology developed in this thesis. The GeoManager was looked at first because it allows multiple

pieces of terrain to register at a common location so that the GeoManager can help objects needing terrain data reach the correct GeoTerrainGrid. Second was the GeoTerrainGrid. This is the workhorse program. The GeoTerrainGrid builds the 3D models of terrain, calculates elevation at arbitrary positions, and calculates rotation vectors for objects to orient them to the terrain. Finally, GeoLocation3 was discussed. This file is basically the interface for objects to place themselves on GeoTerrainGrids. The VRML node allows the user to specify which functions of GeoTerrainGrids the object should use.

IV. EXPLORING FUTURE POSSIBILITIES

A. INTRODUCTION

The code in this thesis represents a starting point for a thorough terrain package for simulation or for terrain visualization and analysis. Further research or future areas of work include several topics. First, reducing the number of polygons used to render the terrain would improve the frame rate of the display and allow rendering larger areas of terrain. Second, line of sight algorithms are looked at because they are critical to determining if objects placed on terrain can see each other. This data is paramount to military simulation since units must be able to see each other to engage each other with direct fire. The next section introduces several possible extensions to this thesis that add useful terrain-related functionality. Then, the idea of terrain servers is introduced so that a robust server holding large amounts of data can provide many clients with the specific pieces of terrain they need. This leads to the next topic, deformable terrain, where events in a simulation change aspects of the terrain. These changes then propagate to the other clients. Finally, adding terrain related features such as bodies of water, roads, vegetation, and buildings is discussed.

B. REDUCING THE NUMBER OF POLYGONS DISPLAYED

Rendering higher resolution terrain such as DTED level 2 terrain requires tremendous numbers of polygons. For example, the sample DTED level 2 code examples in this thesis form grids that are 60 squares across and 60 squares deep. Thus, there are 3,600 squares each of which is broken into two triangles for a total of 7,200 polygons for an area that is roughly 1.8 km by 1.8 km. The multiple grid example uses 16 of these terrain grids for 115,200 polygons for an area that is 4 minutes wide and 4 minutes deep in latitude and longitude. This equates to an area that is a little over 7 km by 7 km. This area is far too small for a military simulation larger than a battle between two companies. If a military theater of operations was 630 km by 630 km, then it would require 350 terrain grids by 350 terrain grids using the same size terrain grids that are in this thesis. That means 122,500 terrain grids yielding 882 million polygons. That is too many polygons for computers today. A good example is the Radeon™ series video cards from ATI. According to the ATI website at www.ati.com, the Radeon™ 9800 Pro can render

approximately 380 million triangles per second. Thus, the card would need about 2.3 seconds to render one frame. The Radeon™ 9600 Pro is a much more affordable card and can only render 162.5 million polygons per second. This card would need about 5.4 seconds to render one frame. Granted, these cards have culling algorithms that eliminate some of these polygons which would improve the rendering time. However, if the scene is built with fewer polygons, then the video card will have less work and the frame rate will improve. Likewise, pushing 882 million polygons to the video card takes a significant amount of I/O time even with the newest Advanced Graphics Port (AGP) buses. Of course, fewer polygons generally results in a lower quality image. The key is balancing image quality and frame rate performance.

Level of detail is an area of continuing research in computer science. As a result, there are several methods of polygon reduction available. The methods fall into two broad categories: continuous level of detail (CLOD) algorithms and static level of detail algorithms. CLOD algorithms currently show the most promise for large scale terrain because the algorithms use one height array and simply vary the number of polygons built from that height array. Static level of detail algorithms require building multiple static models of the terrain before rendering and then swapping out the models when appropriate. This swapping requires a lot of I/O from the computer which can cause pauses in the display and requires more storage space in memory or on disk. Therefore, using static level of detail is typically reserved for rendering complex objects like a vehicle or person. Terrain is simple enough that CLOD algorithms can work well.

The basic concept of the CLOD algorithms is to render the same piece of terrain with fewer polygons while minimizing the impact on the final rendered image. Terrain that is close to the viewpoint needs to use many smaller polygons to show the detail of the terrain. However, distant terrain can use fewer large polygons without impacting the final image significantly. The human eye cannot see as much detail in distant objects as it can in close ones. Therefore, the larger, less detailed polygons used to render distant terrain look natural to the eye. Figure 22 shows a small height array drawn at three different levels of detail in a CLOD fashion. The highest resolution (Figure 22a) uses 32 polygons while the lowest (Figure 22c) uses only 2 polygons. If this height array was far

enough away from the viewpoint that it only covered a few pixels of screen space, then the two images would look the same even though Figure 22c uses much less detail.

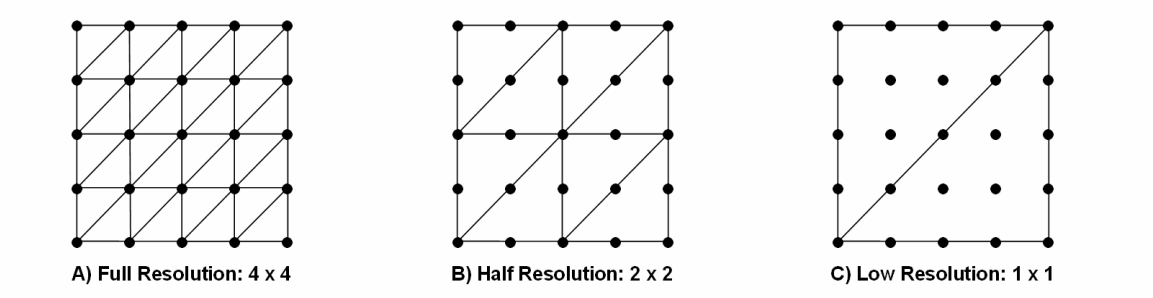

Figure 22. Indexed face set drawn at 3 Resolutions

The screen shot in Figure 23 shows how the GeoTerrainGrids could benefit from some form of polygon reduction by showing a group of GeoTerrainGrids in wire-frame mode. Notice that the near terrain is clearly built from triangles while the distant terrain seems solid. The distant terrain is actually made from triangles that are the same size as the near terrain. However, since the triangles are far away, they project onto a smaller area of the screen and look solid.

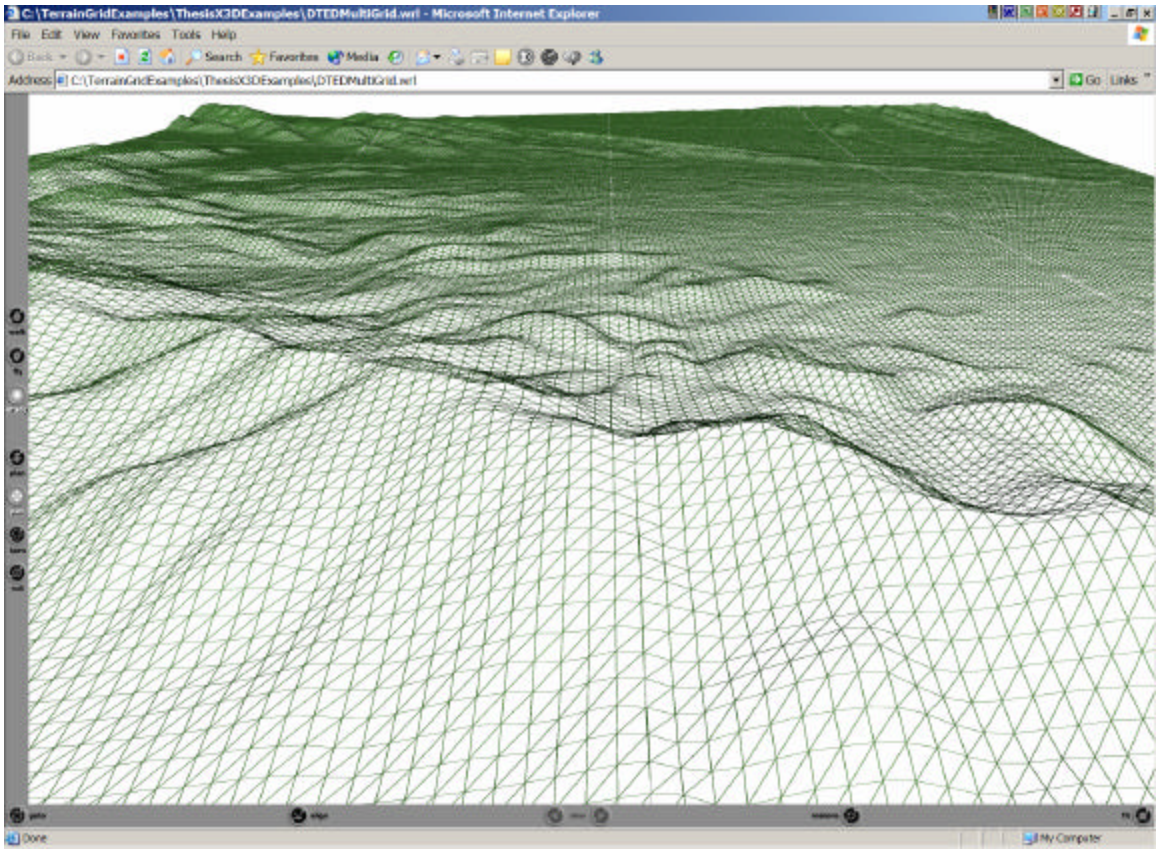

Figure 23. Wire-frame terrain showing how distant terrain polygons do not require polygons as large as near terrain

GeoTerrainGrids are designed so that LOD can easily be added. All of the code for rendering the terrain is contained in the GeoTerrainGrid Java file where it can be studied and modified. A LOD algorithm can be added to explore the effects on image quality and frame rate. In fact, several LOD algorithms can be added to the code and used interchangeably so that they can be compared using the same rendering engine and terrain data. Another benefit of the GeoTerrainGrid approach is that each grid is an independent object that renders itself. This means that every GeoTerrainGrid in the scene can choose its own level of detail for rendering. Distant grids can use low resolution while close up grids can use high resolution. As the viewpoint moves, individual GeoTerrainGrids can change their level of detail independently of the rest. This means that the scene can adjust the LOD of individual sections of the scene as opposed to having to re-compute the whole scene when the viewpoint moves. Programming a

solution for this will require that the GeoTerrainGrids receive the position of the viewpoint. The GeoManager can be expanded to provide this information.

Of course, a LOD algorithm is needed to determine how to build the indexed face set that represents the terrain. A simple distance formula would be a good starting point. The distance from the viewpoint to the GeoTerrainGrid can determine how many polygons to build. This solution can lead to rendering artifacts such as popping as GeoTerrainGrids change LOD and cracking as adjacent GeoTerrainGrids are rendered at different LOD. A more sophisticated solution can calculate the distances that popping will occur at and increase the LOD before reaching that distance. One popular algorithm that addresses this issue is the Real-time Optimally Adapting Meshes (ROAM) algorithm. Basically, this algorithm builds the scene as a triangle bintree. The scene will start as one big right isosceles triangle that is repeatedly partitioned to increase the LOD. The scene can also have triangles merged back together to reduce the LOD as required by the current scene. The algorithm maintains two queues that hold the triangles which can be split and merged respectively. The triangles in these queues are sorted by priority based on what effect the split or merge will have on the scene. As the viewpoint moves, triangles have their priorities updated and are split or merged based on those priorities. When cracks are formed by a split or a merge, the neighbors of the triangle that was just split or merged are visited and likewise split or merged to eliminate the crack. The original algorithm worked well with flight simulators where the viewpoint moved along a path causing the next frame to be similar to the current frame. Therefore, the data in the queues listing the priority of the triangles for splitting and merging was still accurate and only needed some updating. If the viewpoint were suddenly jumped to a new location in the scene, then the ROAM algorithm would have to re-evaluate all the triangles in both queues. Details about an implementation of the ROAM algorithm are available at http://www.llnl.gov/graphics/ROAM/. The ROAM algorithm has since been improved upon. The Stateless One-pass Adaptive Refinement (SOAR) algorithm is a good example. The SOAR algorithm uses the same bintree approach as ROAM, but uses optimized algorithms to quickly rebuild the entire scene using triangle strips for every frame. The triangle strips reduce the amount of data passed to the graphics card through the I/O bus and provide the data in a fairly optimized data format (triangle strips). SOAR

was developed at Lawrence Livermore National Laboratory with details available at http://www.gvu.gatech.edu/people/peter.lindstrom/software/soar/.

C. LINE OF SIGHT (LOS) ALGORITHMS

A line of sight calculation determines what can and cannot be seen from a given position due to terrain obstructing the view. For example, if a person is standing at the bottom of a tall hill and facing the hill, then he cannot see what is on the other side of the hill because the ground is in the way. If this same person moves to the top of the hill, then he will be able to see all sides of the hill at once. Determining what a person or vehicle can see from their current position is paramount to military simulations and military planning. Therefore, if the project explored in this thesis is to move forward as a possible military simulation and planning tool, then adding support for line of sight calculations is critical.

1. Terrain Based LOS Calculations

A line of sight calculation determines if terrain blocks the straight line view between an observer and a target. For simplicity, the observer and the target are typically defined as points in the 3D. This means that when checking for line of sight on an object such as a tank, the tank would be considered to be a point rather than a vehicle that occupies a 3D area. This is a simplification, but if the point is placed at the center top of the tank, then it does reasonably well. A more accurate method could determine a box around the tank and look for visibility with any corner, but that requires 8 line of sight calculations and would not add much realism. To be completely accurate, line of sight calculations would have to cover the entire area the tank occupies. Such a calculation would require Calculus and be computationally intensive. The single point calculation is probably the best compromise of performance and realism.

The line of sight between the observer and the target must then be checked against the terrain. The simplest case happens when both the observer and the target are on the same polygon of terrain. Here, they always have line of sight on each other. However, if they are on different polygons, then multiple checks must be made. First, determine the slope of the target observer line. Then, determine the slope of the line from the observer to several points of terrain on the observer target line. If the slope from the observer to any terrain point on the observer target line is greater than the slope of the entire target

observer line, then there is no line of sight. Basically, if there is high ground between the observer and the target that blocks the view, then the slope of the line to that high ground will be steeper than the slope of the line to the target (Figure 24).

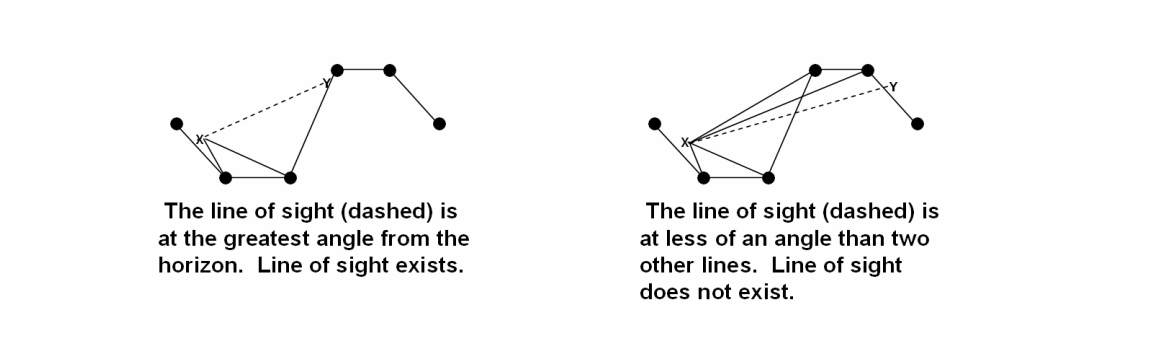

Figure 24. Graphical depiction of determining terrain based line of sight

The next question is how many and what points of terrain to check. A complete solution will check the terrain at the intersection point with every side of every polygon crossed as in Figure 25a. This would require either two or three calculations for every grid square spanned, one when entering the grid square, one when leaving, and possibly one if the line intersects the line that divides the grid square into two triangle polygons. Of course, exiting one grid and entering the next is the same calculation. Therefore, the actual total number of calculations when spanning n grid squares is between n+1 and (2*n) + 1. When objects are many grid squares apart, these calculations will grow quickly. The MODSAF military simulation checks line of sight using this method. Another approach is to always use the same number of calculations and just distribute them evenly across the distance between the observer and target (Figure 25b). This technique could determine that a line of sight exists when it actually does not, but it does keep the number of calculations to a much more manageable number. The Janus military simulation uses this technique. Yet one more technique is to keep the average elevation for every grid square and use that elevation for line of sight calculations. This would mean only one calculation for every grid square spanned. Thus, this is a compromise between the first two methods. Of course, more systems could be devised.

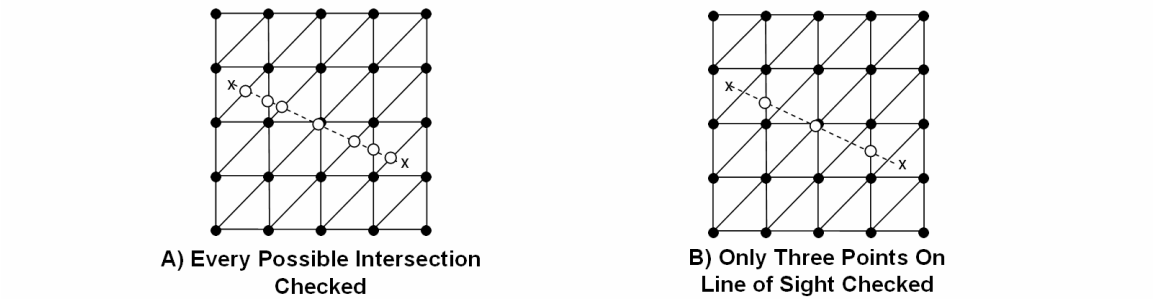

Figure 25. Determining the number of line of sight calculations

2. Horizon Based Line of Sight Calculations

This thesis has been talking about potentially large areas of terrain in true 3D. Thus, the curvature of the earth is a consideration in line of sight calculations. Take, for instance, a line of sight calculation between two ships on the surface of the ocean. Since sea level will have an elevation of zero, these ships would always be able to see each other by the terrain oriented line of sight calculations just described. However, the ships could be far enough apart that the horizon blocks the line of sight. The formula for the location of the horizon is not too difficult, the Pythagorean Theorem is used (Figure 26). Imagine a line from the observer to the horizon and call it r_h for the range to the horizon. This is the first leg of a right triangle. Next, imagine a line going from this point of the horizon to the center of the earth, call it R_e for radius of the earth. This radius is perpendicular to the line r_h. Build a right triangle with these two lines as its legs. The length of the hypotenuse of this triangle is equal to the radius of the earth plus the height of the observer above sea level. The equation looks like this:

$$R_e^2 + r_h^2 = (R_e + h_o)^2$$

Where h_o is the height of the observer above sea level. Solving for r_h gives:

$$r_h^2 = R_e^2 + 2R_e h_o + h_o^2 - R_e^2$$

$$r_h = \mathrm{Sqrt}(2R_e h_o + h_o^2)$$

The radius of the earth (6,374,000 m) is great enough that the h_o^2 term can be insignificant and ignored at low elevations. Any terrain farther away than r_h is beyond the horizon and will not be visible if it is at the surface of the ocean. However, if another object is higher than sea level, then it can still be visible. This range can be calculated.

In fact, if the range from this object to its horizon is calculated in the same fashion as just done above, then the range that the two objects will be able to see each other over the horizon is the sum of their individual r_h values.

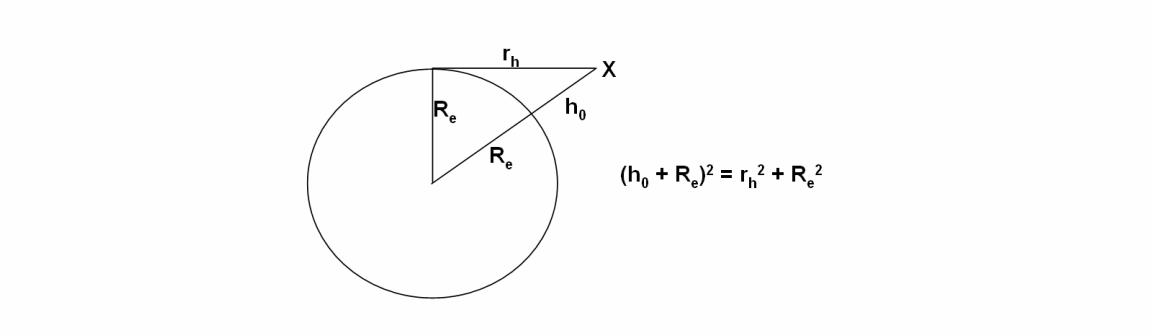

Figure 26. Calculating distance to the horizon

Of course, ground forces will rarely need to worry about forces beyond the horizon. Ground forces will require the terrain oriented line of sight calculations. However, the over the horizon calculations are useful for ships at sea. Likewise, the over the horizon calculation could be used to locate terrain grids that do not have to be rendered at all.

The code for terrain based LOS calculations will be split into two areas. GeoTerrainGrids will determine terrain based LOS within their individual boundaries. Determining which GeoTerrainGrids to use and the points where the LOS enters and/or exits individual GeoTerrainGrids will be done by the GeoManager. Thus, the user will call a method in the GeoManager to determine LOS. The GeoManager will in turn call methods in as many GeoTerrainGrids as necessary.

D. MISCELLANEOUS TERRAIN FUNCTIONS

The last section described LOS functions that could be added to the code in this thesis. Scene graph nodes would access these functions by interacting with the GeoManager class. This same pattern can be used to add even more functions to the code. Three examples are determining the slope of the terrain at a given location, determining the distance between two geographic positions, and determining the straight line path between two positions. Each of these is briefly discussed here.

The slope of the terrain at a given position can affect the speed of a vehicle moving on that terrain. Many vehicles have difficulty climbing steep slopes and slow down when doing so. Likewise, many vehicles travel faster when moving downhill. Finally, some slopes are too steep for vehicles to climb at all. Determining the slope is not too difficult given the position and the direction of travel of the object. Two elevation values are needed. The first must be from the location of the object while the second must be from small distance away in the direction of travel. Determining these elevations was covered in section 3.C.2 of this thesis. The difference between the two elevation values is called the rise of the terrain while the distance between the two locations is called the run. The percentage of the slope is the rise divided by the run multiplied by 100. Figure 27 shows an example calculation using a rise in terrain of 1.5 meters and a run of 4 meters.

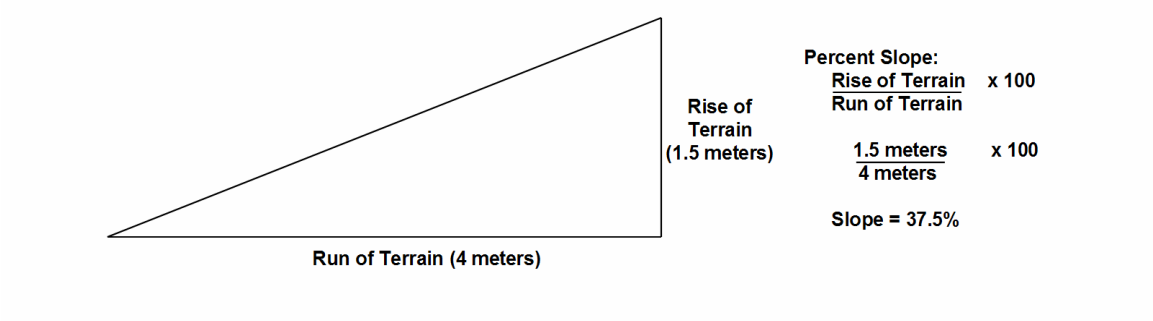

Figure 27. Calculating percent slope

Determining the distance between two geographic positions can be a difficult problem in UTM and latitude and longitude but is much more manageable in geocentric coordinates. Latitude and longitude coordinates do not easily convert to distances because the distance between lines of longitude varies with latitude. UTM coordinates are based on evenly spaced reference lines, but there are still problems at certain locations of the earth. Figure 28 shows a section of the UTM system that demonstrates the problem. The figure is broken down into grid squares that have 2-letter designators. Each grid square is 1 km by 1 km by definition. However, the UTM system divides the world into slices that are 6 degrees of longitude wide. Since the lines of longitude converge at the poles, these 6 degree slices become narrower as they approach the poles. To account for this convergence, grid squares at the boundaries shrink and eventually

disappear while traveling toward the poles. This is visible in the two columns that start with the letters Y and B. At the bottom of Figure 28 these two columns are wider than they are at the top of the figure. If the diagram showed a large enough area, then the two columns would eventually disappear as the poles were approached. These boundaries make distance calculations difficult.

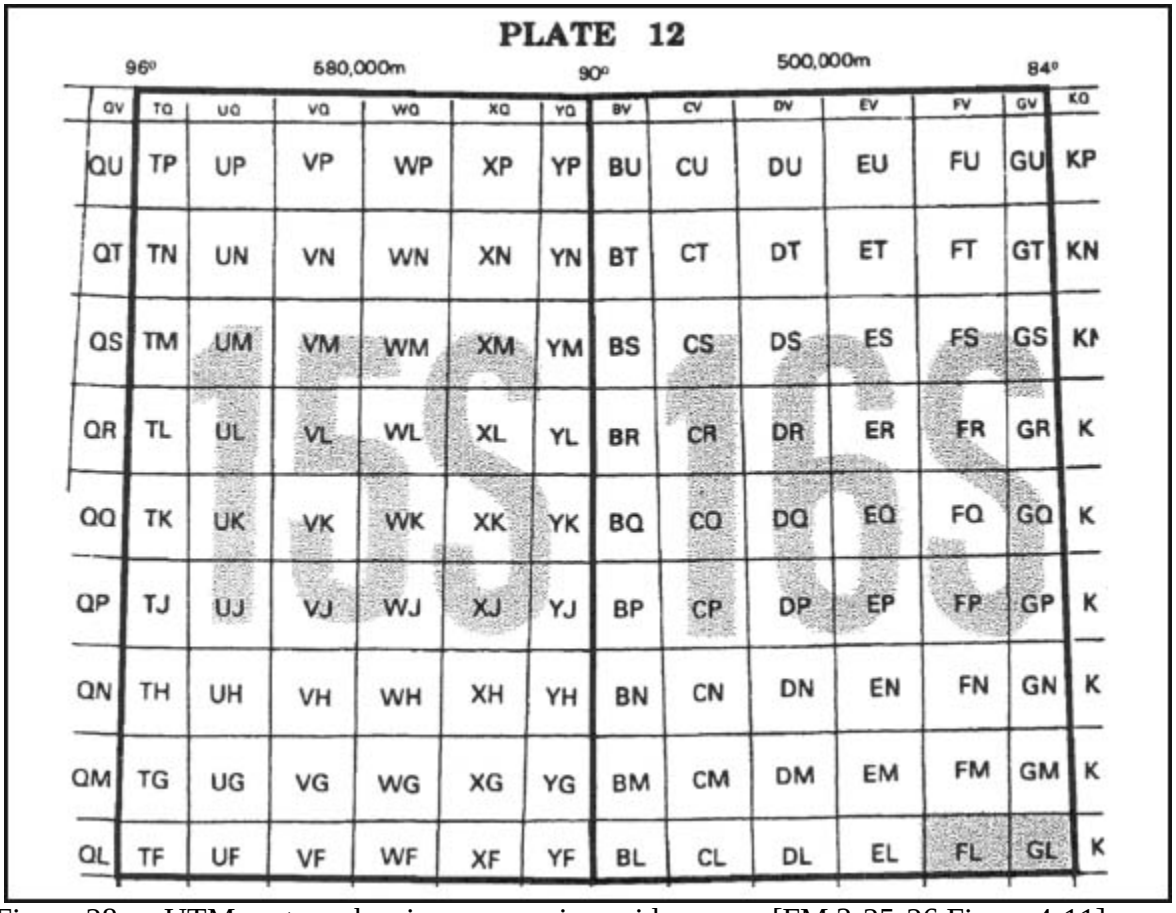

PLATE 12

96° 580,000m 90° 500,000m 84°

QV	TQ	UQ	VQ	WQ	XQ	YQ	BV	CV	DV	EV	FV	GV	KQ
QU	TP	UP	VP	WP	XP	YP	BU	CU	DU	EU	FU	GU	KP
QT	TN	UN	VN	WN	XN	YN	BT	CT	DT	ET	FT	GT	KN
QS	TM	UM	VM	WM	XM	YM	BS	CS	DS	ES	FS	GS	KM
QR	TL	UL	VL	WL	XL	YL	BR	CR	DR	ER	FR	GR	K
QQ	TK	UK	VK	WK	XK	YK	BQ	CQ	DQ	EQ	FQ	GQ	K
QP	TJ	UJ	VJ	WJ	XJ	YJ	BP	CP	DP	EP	FP	GP	K
QN	TH	UH	VH	WH	XH	YH	BN	CN	DN	EN	FN	GN	K
QM	TG	UG	VG	WG	XG	YG	BM	CM	DM	EM	FM	GM	K
QL	TF	UF	VF	WF	XF	YF	BL	CL	DL	EL	FL	GL	K

Figure 28. UTM system showing converging grid squares [FM 3-25-26 Figure 4-11]

Any distance calculation that does not cross one of these 6 degree boundaries is straightforward. The geographic coordinates of the two locations translate directly into the north-south distance between the points and the east-west distance between the points. However, if a 6 degree boundary is crossed, then calculating the distance from the edge of the boundary terrain grid to the boundary itself is difficult. Latitude and longitude coordinates could be helpful here. If the intersection of the 6 degree slice can be computed in latitude and longitude coordinates, then that coordinate can be converted

into a UTM coordinate that may allow the user to determine the distance spanned by that truncated grid square.

The problem is probably best approached using geocentric coordinates. Converting both coordinate locations to GCC gives two sets of X, Y, and Z values that represent a vector from the center of the earth to the coordinate point on the surface. The angle between the two vectors represented by the GCC locations of the two coordinates is needed. The following formula determines this angle.

$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}|\,|\mathbf{b}| \cos ?$

$? = \text{acos}\left((|\mathbf{a}|\,|\mathbf{b}|) / (\mathbf{a} \cdot \mathbf{b})\right)$

Where **a** and **b** are the vectors determined by the GCC values of the two coordinates and · represents the dot product. Of course, this only works when **a** · **b** does not equal zero which only happens when **a** and **b** are orthogonal. In this case, ? is equal to .5p. The distance between the coordinates is then an arc-length problem and the arc-length is simply the radius of the earth multiplied by ?.

The straight-line-path between two locations is a path that, when followed, moves across the terrain while always remaining at the exact elevation of the terrain. A simple straight line between two points will often rise above the terrain or dip below the terrain depending on the specific contours of the terrain between the two points. Building a path that explicitly follows the terrain requires knowing exactly how the terrain was built. With the original GeoElevationGrid determining an explicit path was not possible because the rendering was not explicit (see section 3.C.1 of this thesis). However, with the GeoTerrainGrid, an explicit path can be determined. Implementing an algorithm to determine this path requires determining all the points of intersection with the straight-line path and the triangles in the indexed face set. Figure 29 shows all of these intersections on an example path. The algorithm could store the path in an indexed point set with the points listed in order from the start point to the end point.

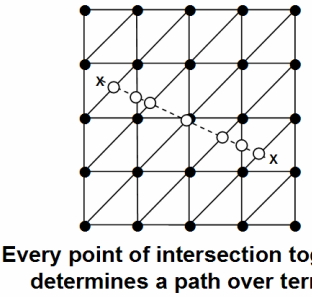

Figure 29. Determining the path over terrain

E. TERRAIN SERVERS

In the second chapter of this thesis, the large amount of data represented by DTED was briefly discussed. Basically, the amount of data is tremendous when higher levels of DTED are used. Therefore, placing the data on every computer that is running as part of a military simulation or training aid is not feasible. However, having a server with robust amounts of storage space that supplies terrain data to clients when needed is feasible. Military simulations are typically run in simulation laboratories with soldiers manning individual workstations that drive simulated forces in the computer. The opposition forces are driven either by more soldiers specifically selected for that task or by civilians. All of the computers participating in the simulation are placed in the same laboratory and are connected to a LAN or WAN. Thus, the simulation is a distributed computer exercise where each computer controls specific icons and listens to the network to keep track of what the other icons on other computers are doing.

The individual computers that soldiers and civilians drive icons from do not have hard drives large enough to store hundreds of gigabytes of terrain data needed to render the entire world at DTED level 2 or higher. One solution to this problem would be to upgrade all of the computers, but that would be far too expensive. A more cost effective solution would be to build networked terrain into the simulations. This way, each computer would only have to store the terrain that the icons it controls are currently occupying. When the icons move to a new area, the software can swap out the old terrain data and replace it with data of their new location. However, this requires a server to handle terrain and networking.

One of the primary concerns with using networked terrain will be bandwidth. Large terrain files require lots of bandwidth to transfer from computer to computer. So, minimizing the amount of data sent is critical. Some possible solutions are multicasting terrain data so that multiple computers will receive data with every transmission and sending terrain data in as compact a format as possible. The multicasting solution requires that multiple computers in the simulation require the same pieces of terrain. This may sound unlikely, but would actually be fairly common. Picture a combat brigade moving toward an objective. There are several types of vehicles being controlled by possibly dozens of individual work stations and all moving toward the same objective. There will be a significant amount of overlap in the terrain needs of those computers. However, a multicasting protocol would be needed that recognizes repeated requests for the same terrain and does not automatically respond to each one.

There is another significant issue with the terrain server concept, though. That issue is providing terrain to several different simulation programs that are all participating in the same distributed simulation (often referred to as a Federation in current military simulations). Each branch of the military has its own simulation programs that have differences. In order to participate in large simulations spanning all the branches, these different programs conform to standards for describing what their icons are doing within the simulation. The current standard is called the High Level Architecture or HLA. Details about HLA are at https://www.dmso.mil/public/transition/hla/. The actual specification is the IEEE 1516 specification and is available for purchase at http://shop.ieee.org/store/product.asp?prodno=SS94883. A standard similar to HLA will be needed for networked terrain. One possibility is to send DTED files over the network. However, DTED files are in a binary format that can be difficult to read and can be inflexible in the amounts of data transferred. Another solution is using XML to transfer data. XML was described in Chapter II of this thesis and provides a more elegant solution where the user could use XML tools to draw the data out of the files without having to meticulously parse a binary file format. Likewise, using XML allows validating terrain data files to ensure that every program on the system is producing proper terrain files. The XML version of a DTED file would be significantly larger than the original file, increasing the bandwidth needed. However, XML files are text based

and work well with common compression algorithms. Using such algorithms can bring the size of the files back down to save some of this bandwidth. Ekrem Serin of the Naval Postgraduate School did a thesis that addresses serializing binary objects created from XML [Serin, 2003]. His thesis is located at theses.nps.navy.mil/03Mar_Serin.pdf and presents ideas that can be used to create a robust networked terrain solution that uses open standards.

F. DEFORMABLE TERRAIN

Having deformable terrain means that actions within a simulation can change the shape of the terrain with persistent effects. For terrain discussed so far in this thesis, this would result in changing the height values of the underlying DTED data. These changes would then be propagated to all other clients when the particular area of terrain is used. How the terrain is modified could vary from digging a tank ditch, to filling in a culvert, to blowing large holes in the earth with bunker busting thousand pound bombs. The code to determine how these occurrences affect terrain is not a subject for this thesis. What is relevant here are techniques to propagate those changes throughout a distributed military simulation.

One possible technique is to store complete terrain files on every computer in the simulation. When one computer determines that a terrain altering event has occurred, the change in terrain is calculated and sent out to every computer in the distributed system. Sending the entire terrain file every time a small change occurs would be terribly inefficient. Therefore, a protocol for sending small changes would be used. If this protocol was a reliable protocol, then the terrain change would only be sent as many times as necessary for all the systems in the simulation to acknowledge receipt. Each system would then store the change on local storage in case the system required rebooting at some time. The problem with this solution is that if a system goes off-line, then that system will not acknowledge receipt of the terrain change and will cause repeated re-transmissions until the system comes back on-line and acknowledges receipt. An unreliable multicasting protocol could be used instead. However, with this situation, the change in terrain would have to be retransmitted at regular intervals for the duration of the simulation because there is no mechanism to verify that everyone has received and understood the message. This periodic retransmission is called a heartbeat and insures

that eventually, every computer listening to the system will get the message. This technique will not scale well at all, though. Deformed pieces of terrain would use up bandwidth retransmitting the deformation until the exercise ended. This technique will not work in large simulations.

A second approach involves the terrain server. This system needs a separate networking solution for passing terrain to clients because some form of reliability is required. In other words, the terrain server cannot use the heartbeat method for passing terrain data. There is too much data for that and it rarely changes. Deformable terrain would require reliable networking in both directions. This does not mean that TCP/IP must be used like it is with most reliable networking solutions. In fact, TCP/IP is not a multicasting protocol. Therefore, some other reliable protocol that supports multicasting would be desirable. Unfortunately, there are no common or proven networking protocols that support reliable multicasting. One possible future solution is the Selectively Reliable Multicast Protocol (SRMP) being developed at George Mason University by Mark Pullen [Shanmugan, 2002]. Details about the system are at netlab.gmu.edu/SRMP/contact.php. However, some investigating and testing will be required. Terrain altering events would be sent to the terrain server(s). From there, clients could be notified that a change has occurred in a specific area. If a system joins late or has to restart, then it will automatically contact the server for its terrain data and will receive the new deformed terrain. The toughest question is how to inform all of the clients of the change. Some clients will need to update their terrain data while others will not be interested in that terrain at that time. So, the question is whether the server somehow keeps track of which clients are currently using the terrain that was just deformed or whether the server bothers every client including those that do not have that terrain loaded at the time. Personally, the author thinks sending an update notification to every client is the better solution because terrain changes are rare enough that forcing the server to keep track of what terrain each client currently has loaded is not justified. In fact, it is possible that the two way communication that keeping track of every client's terrain use would require could be more burdensome on both the server and the client than just demanding a negative or positive response to a terrain change from every client.

G. GEOGRAPHIC FEATURES SUCH AS BODIES OF WATER, ROADS, VEGETATION, AND BUILDINGS

One last topic of discussion is how to include terrain features that are not present in a height field. For instance, roads are not present in DTED. However, roads are very significant in military simulations. The question is how can a road be represented within the simulation so that it can be utilized as a road by objects in the simulation, rendered as a road, and possibly deformed by events? The problem is that an item such as a road is neither an object like the icons that represent vehicles and personnel, nor is it ordinary terrain. For instance, roads cannot be rendered with the terrain engine in this thesis because they do not conform to evenly spaced postings in a height field. Additional code specifically aimed at rendering a road would have to be written. Likewise, the path of the road along with the dimensions and type would have to be available to determine how to render the road. After all, a gravel road looks significantly different than an interstate highway.

Geographic features such as roads, bodies of water, and buildings will have to be stored as objects. In fact, each type of geographic feature will require its own type of object, possibly multiple types of objects. Take for instance, the road example again. A road object would be able to store information about the route the road follows, the width of the road, what type of texture map to use when rendering the road, how the road affects trafficability, and more. Likewise, the road object will have code to draw the road in the rendering engine. Complicated simulations could even have code that degraded the road every time a vehicle passed over it. Naturally, the road will have to interact with the terrain objects for the display to work properly. The road will need to access height field data in order for the road to sit properly on the surface of the terrain at all locations. Simply retrieving the proper terrain elevation at a point will not work because the road will span many polygons of terrain. Instead, the road will have to know how the terrain polygons are being built so that it can match those polygons. If the road simply takes some spot elevation, then there will be places where the road floats above the terrain a little and places where it is buried a little.

The next question is how to control and distribute these objects throughout the system. Normally, objects are controlled by clients who routinely update the system with

the status of the objects using heartbeat packets. However, if every road, building, body of water, and tree suddenly needed to give off heartbeat packets, then the network would become saturated. The terrain related objects could have their heartbeat packets sent much less frequently and at spaced intervals, but they would still be consuming bandwidth and client processor cycles for handling the packets. Another solution is to treat the objects just like terrain. The terrain server sends the objects along with the terrain. This would likewise help the clients to associate the terrain objects with the appropriate terrain grids so that when a terrain grid is rendered at a lower resolution because it is far away from the viewpoint, the terrain objects can match that lower resolution. The terrain objects could even be given direct access to the height field and indexed face sets to assist in the rendering process.

Another benefit of treating the geographic features similarly to terrain is that the terrain server owns the objects. Everything in the simulation is owned by some computer in the simulation. If that computer goes offline, then those objects disappear from the simulation until that computer comes back on-line. If the geographic objects gave off heartbeats, then they would disappear from the system anytime their computer goes off-line long enough. The terrain itself, though, would not disappear even if the terrain server went off-line. This is because the terrain is persistent until the client is told there is a change. Of course, none of the clients will be able to retrieve any additional pieces of terrain nor will they be able to report any terrain deformations while the server is off-line, but the simulation would be able to continue.

Here are some final thoughts on many of these types of objects. The objects will need the necessary code to render themselves. This code would ideally allow the objects to render at multiple levels of detail that mirrors the level of detail that the terrain around them is rendering itself at. The objects will also need code to handle how simulation objects interact with them. For instance, vehicles will typically move faster on roads and will sink in water. This will require the ability to communicate with the simulation objects. Each object will also need code to determine how it is deformed by various actions. A tank round will not know how to deform a road versus a building versus a body of water. Instead, the geographic objects will have to contain this information. Granted, the objects will have broad categories such as explosive projectile or kinetic

energy round, but the code for calculating damage will have to be in the geographic objects. The munitions object will only contain data describing the weapon. These geographic objects will also need to be robust enough to understand how deformations affect how the object renders itself and how it interacts with simulation objects. Some of the simpler geographic objects will be roads and bodies of water. After all, they could simply be viewed as improvements to mobility and restrictions to mobility. Vegetation is tougher because it also affects visibility. Line of sight algorithms would have to take vegetation into consideration. Thicker vegetation would limit the line of sight and lower the probability of detecting simulation objects within it. Buildings would be very difficult. Here, simulation objects could enter the building and interact with it. Objects within the building would not be visible from the outside unless they are near a window, door, or hole. Buildings can be damaged so that the appearance changes significantly. The change may be a scorch mark, a small hole, or maybe even a collapsed wall. The line of sight algorithms for terrain will not work because the buildings have walls that have windows and doors. Even if damaging buildings is not allowed, simply determining an efficient way to define the structure and appearance of a building is complicated.

Lastly, one of the most difficult problems will be getting the majority of the simulation programs being used to agree upon how to implement these geographic objects. The various simulation programs will already be handling terrain in a similar manner, but these objects will be very different. Air Force and Navy simulations will probably treat objects such as roads simply as visual objects for display. However, Army and Marine simulations will treat roads as both visual objects for display and terrain objects that impact mobility. Air Force simulations will have situations where planes are at high altitude and capable of viewing hundreds or even thousands of buildings at one time. If each of these buildings is a large and complicated object, then the Air Force simulation will not be able to load the scene efficiently without sophisticated culling and level of detail algorithms. However, the Army and Marines will need the buildings to be robust objects that the icons of soldiers can enter and interact with. Meeting both of these needs simultaneously so that the Air Force, Army, and Marines can run a joint simulation will be a difficult task.

H. SUMMARY

This chapter focused on five possible extensions to the technology developed in this thesis. The first topic, polygon reduction, would improve the frame rate of terrain displays and increase the maximum amount of terrain that could be viewed at one time. The second topic was line of sight algorithms. Two algorithms were discussed, one for local terrain and a second for over-the-horizon calculations. These algorithms are used extensively by military simulations. The next topic was terrain servers. Most computers today do not have the space to store terrain data for the entire planet. Therefore, a networking solution where robust servers hold all the terrain for the world and give smaller client computers the specific pieces of terrain that they need would allow smaller computers to have full access to data on the entire planet. The fourth topic, deformable terrain, would allow simulations to explore events that change the surface of the earth. Finally, some of the issues that will be encountered when trying to incorporate terrain related objects into the scene are addressed. These items include things like roads and vegetation that are not actually terrain, but that stay geographically fixed.

APPENDIX A. ACRONYMNS AND ABBREVIATIONS

AGP	Advanced Graphics Port
CLOD	Continuous Level Of Detail
DTED	Digital Terrain Elevation Data
GCC	Geocentric Coordinate
GDC	Geodetic Coordinate
GeoVRML	Geographic Virtual Reality Modeling Language
LAN	Local Area Network
LOD	Level Of Detail
LOS	Line Of Sight
ROAM	Real-Time Optimally Adapting Mesh
SAVAGE	Scenario Authoring and Visualization for Advanced Graphical Environments
SOAR	Stateless One-pass Adaptive Refinement
SRMP	Selectively Reliable Multicast Protocol
UTM	Universal Transverse Mercator
VRML	Virtual Reality Modeling Language
WAN	Wide Area Network
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language for Transformations
X3D	Extensible 3D Graphics

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. CODE EXAMPLES

The code for this thesis started out as several files, but was condensed into only three. All the code is written in Java so that it can work with VRML as part of a script node. The first Java class is the GeoManager. This is the most unique class of the three because it does not extend the Script class. This means that it is not associated with a X3D or VRML node. The GeoManager is a helper class that allows the GeoLocation3 class and the GeoTerrainGrid class to work together and extend the power of X3D and GeoVRML

A. GEOMANAGER

The key to this class is the private constructor and the static method to retrieve a copy of a GeoManager. Together, these aspects ensure that only one GeoManager exists at any time and that all GeoTerrainGrids and GeoLocation3s can work together by communicating through this single GeoManager. The code is commented extensively to assist the reader in understanding the code.

```
import java.util.*;
import geotransform.coords.Gdc_Coord_3d;

public class GeoManager
{
  // One static manager is defined - this is the only
manager that anyone
  //  will be allowed to use (GeoTerrainGrids and
GeoEntities)
  private static GeoManager manager = null;

  // This Vector holds all the terrain grids that register.
Remember that when
  //  coding in Java, only Java 1.1 methods can be used.
This significantly
  //  limits the methods available for Vectors.
  protected Vector grids;

  // This vector holds all the entities that register.
  protected Vector entities;
```

```
  // This hashtable holds every URL received and
automatically does not
  //  record duplicate entries
  protected Hashtable urlKeeper;

  // Turn this to false to shut off debugging statements to
the console
  private boolean debug = true;

  // The only way to access the GeoTerrainManager is to
call this static method
  public static synchronized GeoManager getGeoManager()
  {
    // Check to see if a manager has already been created
    if(manager == null)
    {
      // None has, so create one
      manager = new GeoManager();
    }
    // Return the static manager - it may have just been
created
    return manager;
  }

  /**
  *  Constructor; should not be called by external people.
To get
  * an instance of this class, call getTerrainManager
  */
  private GeoManager()
  {
    debugOut("Inside constructor for GeoManager.  Manager =
" + this);
    grids = new Vector();
    entities = new Vector();
    urlKeeper = new Hashtable();
  }

  /**
   * Add a new grid to the list
   */
  public void addGrid(GeoTerrainGrid newGrid)
  {
    grids.addElement(newGrid);
    debugOut("Adding new GeoTerrainGrid.  Total number of
grids: " + grids.size());
  }
```

```
  public GeoTerrainGrid getGrid(Gdc_Coord_3d coordinate)
  {
    GeoTerrainGrid tempTerrainGrid = null;
    boolean found = false;

    for(int x = 0; x < grids.size(); x++)
    {
      tempTerrainGrid = (GeoTerrainGrid)grids.elementAt(x);
      if(tempTerrainGrid.checkBounds(coordinate))
      {
        found = true;
        break;
      }
    }
    if(found)
    {
      return tempTerrainGrid;
    }
    else
    {
      debugOut("Did not find a grid");
      return null;
    }
  }

  private void debugOut(String message)
  {
    if(debug)
      System.out.println(message);
  }
}
```

B. GEOTERRAINGRID NODE

The GeoTerrainGrid is the workhorse of this thesis. This code creates the indexed face set that renders the terrain, calculates the elevation at arbitrary positions, and determining the rotation vector and angle needed to orient an object to the terrain under it. There is license agreement attached to this code because it is an extension to the GeoElevtionGrid from SRI. Therefore, the license agreement from SRI is included. All changes to the original GeoElevationGrid have been clearly marked.

```
//
// Filename: GeoTerrainGrid.java (formerly
GeoElevationGrid.java)
//
// Author:
//   Martin Reddy, SRI International.
//   Heiko Grussbach, Centre Recherche Henri Tudor
//   Yong-Tze Chi, SRI International
//   CPT Brian Hittner, US Army, Naval Postgraduate School
student
//
// Purpose:
//   This class implements a new ElevationGrid node for
VRML. It enables the
//   specification of coordinates in coordinate systems
other than the
//   basic VRML Cartesian XYZ system. We support a number
of geographic
//   coordinate systems such as lat/long and UTM.
//
//   This code requires access to the GeoTransform Java
package, included
//   as part of the GeoVRML source code distribution.
//
// License:
//   The contents of this file are subject to GeoVRML
Public License
//   Version 1.0 (the "License"); you may not use this file
except in
//   compliance with the License. You may obtain a copy of
the License at
//   http://www.geovrml.org/1.0/license/.
//
//   Software distributed under the License is distributed
on an "AS
//   IS" basis, WITHOUT WARRANTY OF ANY KIND, either
express or
```

```
//   implied. See the License for the specific language
governing
//   rights and limitations under the License.
//
//   Portions are Copyright (c) SRI International, 2000.
//
// Revision:
//   Martin Reddy: initial version
//   Heiko Grussbach (28 Feb 2000): optimized conversion to
GCC
//   Yong-Tze Chi (20 Jul 2000): get round get1Value()
error in Cosmo
//   Brian Hittner (26 Sep 2003): Major revision
//      - Added ability to register grid with a GeoManager
//      - Added ability to retrieve correct elevation to
place an object on
//         the ground at a given location
//      - Added ability to retieve correct rotation to
place an object on
//         the ground at a given location
//      - Replaced maintaining the indexed face set,
coordinate list, and height
//         field at class level with retrieving same values
from the VRML Node
//         when needed to reduce the memory footprint
//
//   $Id: GeoElevationGrid.java,v 1.2 2002/03/08 00:30:25
reddy Exp $
//

import java.lang.*;
import vrml.*;
import vrml.field.*;
import vrml.node.*;
import geotransform.coords.*;
import org.web3d.geovrml.GeoVRML;

// ************* Start CPT Brian Hittner Addition Sep 2003
**************
import geotransform.transforms.*;
import java.util.*;
// ************* End CPT Brian Hittner Addition Sep 2003
**************

public class GeoTerrainGrid extends Script {

  GeoVRML geovrml = null;
```

```
  double  yScale;
  double  xInc, zInc;
  int     xDimension, zDimension;
  boolean debug = false;

  // ************* Start CPT Brian Hittner Addition Sep
2003 **************
  /** The following variables are no longer kept at the
class level.  Instead, the
   * values are retrieved at run-time from the underlying
VRML object to reduce
   * the memory footprint of this class.
   */
//  Node    ifs, coord;
//  MFFloat height = null;
  // A validHeightField boolean is used so that the code
does not try to calculate
  //  anything if the height field was initially invalid or
later set to something
  //  invalid.  It is set to true only once a valid height
field is processed.
  boolean validHeightField = false;
  Gcc_Coord_3d gccGeoOrigin; // This is needed to convert
coordinates that are
                             //  in screen coordinates back
to GCC space
  // The TerrainManager keeps track of terrain grids by
their boundaries.  The
  //  boundaries are stored as the south west corner and
the north east corner.
  Gdc_Coord_3d gdcSouthWest, gdcNorthEast;
  Utm_Coord_3d utmSouthWest, utmNorthEast;
  // The GeoManager allows GeoLocation3 to locate the
proper terrain grid
  GeoManager manager;
  // ************* End CPT Brian Hittner Addition Sep 2003
**************

  String  geo_system;
  String  geoGridOrigin;

  // regenerate() will build the vertex lists based upon
the current
  // GeoElevationGrid state, e.g. yScale, height array,
etc.
  // Note: parameter height was added by CPT Brian Hittner,
Sep 2003
```

```
  private void regenerate(MFFloat height) {
    // ************* Start CPT Brian Hittner Addition Sep
2003 **************
    // These VRML Nodes are needed to build the
IndexedFaceSet that is the
    //  rendered terrain (These used to be class variables)
    Node ifs;
    Node coord;

    // Retrieve the Nodes that define the IndexedFaceSet
    coord = (Node)((SFNode)getField("coord")).getValue();
    ifs   = (Node)((SFNode)getField("ifs")).getValue();
    // ************* End CPT Brian Hittner Addition Sep
2003 **************

    // get the texCoord coordinate list. If vrml_texpoint
is non-null
    // after this, then we need to generate texture
coordinates

    float h[];

    float vrml_texpoint[] = null;
    Node texCoordNode = null;
    MFVec2f tex_point = null;

    SFNode texCoord = (SFNode) ifs.getExposedField(
"texCoord" );
    if ( texCoord != null ) {
      texCoordNode = (Node) texCoord.getValue();
      if ( texCoordNode != null ) {
     tex_point = (MFVec2f) texCoordNode.getExposedField(
"point" );
     if ( tex_point.getSize() == 0 )
       vrml_texpoint = new float[ xDimension * zDimension *
2 ];
      }
    }

    // let's allocate an array to hold all of the (x,y,z)
coords

    float vrml_point[] = new float[ xDimension * zDimension
* 3 ];

    // loop through all height field values
```

```
    int h_index = 0, p_index = 0, t_index = 0;

    //Insertion starts Heiko Grussbach
    double[] geoGridOriginArray=new double[3];
    double[] new_coord=new double[3];

    geoGridOriginArray=geovrml.getValues(geoGridOrigin,3);
    float xDiv=1.0f/( xDimension - 1.0f );
    float zDiv=1.0f/( zDimension - 1.0f );
    //Insertion ends Heiko Grussbach

    h = new float[zDimension * xDimension];
    height.getValue(h);

    for ( int z = 0; z < zDimension; z++ ) {
      for ( int x = 0; x < xDimension; x++ ) {

     // get this elevation value (implement vertical
exaggeration here)

     // double elev = (double) height.get1Value( h_index++
) * yScale;
        double elev = (double) h[h_index++] * yScale;

     // work out the string describing this new geographic
location

     //Change starts Heiko Grussbach
     geovrml.addCoord( new_coord, geoGridOriginArray,
xInc*x, zInc*z,
                elev, geo_system );
     //Change ends Heiko Grussbach

     if ( new_coord == null ) return;

     // convert this into GCC

     Gcc_Coord_3d gcc = geovrml.getCoord( new_coord,
geo_system );

     // and then add it to our list of floats

     vrml_point[p_index++] = (float) ( gcc.x ); // /
1000000.0 );
     vrml_point[p_index++] = (float) ( gcc.y ); // /
1000000.0 );
```

```
     vrml_point[p_index++] = (float) ( gcc.z ); // /
1000000.0 );

     // and update our texture coordinate list too

     if ( vrml_texpoint != null ) {
       //Change starts, Heiko Grussbach
       vrml_texpoint[t_index++] = (float) x *xDiv;
       vrml_texpoint[t_index++] = (float) z *zDiv;
       //Change ends, Heiko Grussbach
     }

     if ( debug )
       System.out.println( h_index + ": " +
                     vrml_point[p_index-3] + ", " +
                     vrml_point[p_index-2] + ", " +
                     vrml_point[p_index-1] + " : " +
new_coord );
      }
    }

    h = null;

    // Now let's make the coords field of our Coordinate
node
    // equal to the list of coordinates that we have just
built

    MFVec3f coord_point = (MFVec3f) coord.getExposedField(
"point" );
    coord_point.setValue( p_index, vrml_point );

    // set the texCoord field if we are generating texture
coords

    if ( vrml_texpoint != null && tex_point != null ) {
      tex_point.setValue( t_index, vrml_texpoint );
    }

    // let's make the coordIndex entries. These are a bit
easier!

    int values[] = new int[ ( xDimension -1 ) * (
zDimension - 1 ) * 8 ];

    int index = 0;
```

```
    for ( int z = 0; z < zDimension - 1; z++ ) {
      for ( int x = 0; x < xDimension - 1; x++ ) {
        // ************* Start CPT Brian Hittner Addition
Sep 2003 **************
        // This code was modified to build 2 triangles per
grid square instead
        //  of one complete square that the rendering
engine would then have
        //  to split into triangles at run-time
     values[index]   = x + z * xDimension;
     values[index+1] = (x + 1) + (z + 1)*xDimension;
     values[index+2] = x + (z + 1)*xDimension;
     values[index+3] = -1;
     values[index+4] = x + z*xDimension;
        values[index+5] = (x + 1) + z * xDimension;
        values[index+6] = (x + 1) + (z + 1)*xDimension;
        values[index+7] = -1;
        // ************* End CPT Brian Hittner Addition Sep
2003 **************

     if ( debug )
       System.out.println( "Poly: " + values[index] + " "
+values[index+1] +
                           " " + values[index+2] + " " +
values[index+3] +
                           "  (" + x + "," + z + ": " + index +
")" );
     index += 8;
      }
    }

    MFInt32 coord_index = (MFInt32) ifs.getEventIn(
"set_coordIndex" );
    coord_index.setValue( index, values );

    // ************* Start CPT Brian Hittner Addition Sep
2003 **************
    validHeightField = true;
    // ************* End CPT Brian Hittner Addition Sep
2003 **************
    // we're done!

    if ( debug )
      System.out.print( "GeoElevationGrid: done." );

  }
```

```
  // processEvent deals with all of the eventIns that we
support.
  // Currently this includes set_yScale and set_height

  public void processEvent( Event e ) {

    // ************* Start CPT Brian Hittner Addition Sep
2003 **************
    MFFloat height;
    // ************* End CPT Brian Hittner Addition Sep
2003 **************
    if ( debug ) System.out.println( "Event received: " +
e.getName() );

    // set_yScale lets you change the vertical exaggeration
on the fly

    if ( e.getName().equals( "set_yScale" ) ) {
      ConstSFFloat value = (ConstSFFloat) e.getValue();
      yScale = value.getValue();
      // ************* Start CPT Brian Hittner Addition Sep
2003 **************
      height = (MFFloat)getField("height");
      if(height.getSize() > 1)
        regenerate(height);
      else
        validHeightField = false;
      // ************* End CPT Brian Hittner Addition Sep
2003 **************
    }

    // set_height lets you change the height values on the
fly

    if ( e.getName().equals( "set_height" ) ) {
      ConstMFFloat cmffloat = (ConstMFFloat) e.getValue();
      float values[] = new float[cmffloat.getSize()];
      cmffloat.getValue( values );
      height = new MFFloat( values );
      // ************* Start CPT Brian Hittner Addition Sep
2003 **************
      if(height.getSize()>1)
        regenerate(height);
      else
      {
        // a null height field was received, clear the
indexed face set
```

```
        Node indexedFaceSet, coord;
        float[] array    = new float[0];
        int[]   intArray = new int[0];
        int     index    = 0;
        // Retrieve the Nodes that define the
IndexedFaceSet
        coord          =
(Node)((SFNode)getField("coord")).getValue();
        indexedFaceSet =
(Node)((SFNode)getField("ifs")).getValue();
        MFVec3f coord_point =
(MFVec3f)coord.getExposedField("point");
        coord_point.setValue(index, array);
        MFInt32 coord_index =
(MFInt32)indexedFaceSet.getEventIn("set_coordIndex");
        coord_index.setValue(index, intArray);
        validHeightField = false;
      }
      // ************* End CPT Brian Hittner Addition Sep
2003 **************
    }

  }

  // The initialize method is called when the Node is first
loaded.
  // Here we grab copies of any necessary
fields/eventIn/eventOuts

  public void initialize() {

    // Take copies of all the fields for this node

    SFNode   geoOrigin     = (SFNode) getField( "geoOrigin"
);
    MFString geoSystem     = (MFString) getField(
"geoSystem" );

    SFString xSpacing      = (SFString) getField(
"xSpacing" );
    SFString zSpacing      = (SFString) getField(
"zSpacing" );

    // ************* Start CPT Brian Hittner Addition Sep
2003 **************
    MFFloat height; // This used to be a class variable
```

```
    // ************* End CPT Brian Hittner Addition Sep
2003 **************

    geoGridOrigin = ((SFString)
getField("geoGridOrigin")).getValue();
    height        = (MFFloat) getField( "height" );
    xDimension    = ((SFInt32) getField( "xDimension"
)).getValue();
    zDimension    = ((SFInt32) getField( "zDimension"
)).getValue();
    yScale        = (double)((SFFloat) getField( "yScale"
)).getValue();

    // ************* Start CPT Brian Hittner Addition Sep
2003 **************
    // These values are retrieved inside the regenerate
method now
//    coord         = (Node) ((SFNode) getField( "coord"
)).getValue();
//    ifs           = (Node) ((SFNode) getField( "ifs"
)).getValue();
    // ************* End CPT Brian Hittner Addition Sep
2003 **************

    debug         = ((SFBool) getField( "debug"
)).getValue();
debug = true;

    // convert the spacing strings into double values

    xInc =
(Double.valueOf(xSpacing.getValue())).doubleValue();
    zInc =
(Double.valueOf(zSpacing.getValue())).doubleValue();

    // ready to start...

    if ( debug )
      System.out.println( "GeoElevationGrid: " + xDimension
+ " x " +
                  zDimension + "(" + xInc + ":" + zInc + ")"
);

    // do some sanity checks

    if ( xDimension < 2 || zDimension < 2 ) {
```

```
      System.out.println( "xDimension and zDimension must
be >= 2" );
      return;
    }

    // Okay, let's initialise the GeoVRML utility class
    // These classes should be installed on the user's
system and in
    // their CLASSPATH. If they are not, then we can't do
anything!

    try {
      geovrml = new GeoVRML();
    } catch ( NoClassDefFoundError e ) {
      System.out.println( "GeoTransform classes not
installed in CLASSPATH!" );
      return;
    }

    geovrml.setOrigin( geoOrigin );
    geo_system = geovrml.VRMLToString( geoSystem );

    // build the IndexedFaceSet from the GeoElevationGrid
data
    // ************* Start CPT Brian Hittner Addition Sep
2003 **************
    // The geoOrigin is stored as a GCC so that coordinate
points can be
    //  converted between GCC space and render frame space
    gccGeoOrigin = geovrml.getOrigin();
    // The addCord() method used here will determine the
north east corner of
    //  the grid based on the dimension of the grid and the
geoSystem
    String tempNorthEast = geovrml.addCoord(geoGridOrigin,
xInc*(xDimension-1),
                                        zInc*(zDimension-1),
0.0, geo_system);
    if(geo_system.startsWith("UTM"))
    {
      // When the TerrainGrid is UTM based, both GDC and
UTM coordinates for
      //  the boundaries are needed because checkBounds()
only uses GDC
      gdcSouthWest = convertUtmToGdc(geoGridOrigin);
      gdcNorthEast = convertUtmToGdc(tempNorthEast);
      utmSouthWest = new Utm_Coord_3d();
```

```
      utmNorthEast = new Utm_Coord_3d();
      Gdc_To_Utm_Converter.Init();
      Gdc_To_Utm_Converter.Convert(gdcSouthWest,
utmSouthWest);
      Gdc_To_Utm_Converter.Convert(gdcNorthEast,
utmNorthEast);
    }
    else
    {
      // For GDC TerrainGrids, only GDC coordinates are
needed for the boundaries
      gdcSouthWest = parseGDC(geoGridOrigin);
      gdcNorthEast = parseGDC(tempNorthEast);
      utmSouthWest = null;
      utmNorthEast = null;
    }

    // This terrain grid must register with the terrain
manager so that entities
    //  can reference this terrain grid for positioning
data later
    manager = GeoManager.getGeoManager();
    manager.addGrid(this);
    regenerate(height); // Added the parameter since the
height field is now
                        //  a local variable instead of a
class variable
 }

 // Takes a gdcCoordinate as a String and turns it into a
Gdc_Coord_3d
 private Gdc_Coord_3d parseGDC(String gdcCoordinate)
 {
   Gdc_Coord_3d coord;
   double[] array = new double[3];
   StringTokenizer tokenizer = new
StringTokenizer(gdcCoordinate, " ");
   for(int i = 0; i <= 2; i++)
     array[i] = new
Double(tokenizer.nextToken()).doubleValue();
   coord = new Gdc_Coord_3d(array[0], array[1], array[2]);
   return coord;
 }

 // Warning: this routine only works if the current
geoSystemString is set
 //  to UTM with a zone included
```

```
 private Gdc_Coord_3d convertUtmToGdc(String utm)
 {
   Gcc_Coord_3d tempGcc = geovrml.getCoord(utm,
geo_system);
   float[] floatGridArray = new float[3];
   floatGridArray[0] = new Double(tempGcc.x).floatValue();
   floatGridArray[1] = new Double(tempGcc.y).floatValue();
   floatGridArray[2] = new Double(tempGcc.z).floatValue();
   String tempOrigin = geovrml.geoCoord(floatGridArray,
"GD");
   // tempOrigin is now in GD coordinate space (lat/long)
as a String
   Gdc_Coord_3d tempGdc = parseGDC(tempOrigin);
   return tempGdc;
 }

 // receives a location and checks to see if that location
is within the area
 //  covered by this particular GeoTerrainGrid
 public boolean checkBounds(Gdc_Coord_3d location)
 {
   if(validHeightField)
   {
     if(location.latitude >= gdcSouthWest.latitude &&
        location.latitude <= gdcNorthEast.latitude &&
        location.longitude >= gdcSouthWest.longitude &&
        location.longitude <= gdcNorthEast.longitude)
       return true;
     else
       return false;
   }
   else
     return false;
 }

 // receives a location and calculates the proper rotation
that will rotate an
 //  object to coincide with the normal of the ground at
that location
 public SFRotation getOrientation(Gdc_Coord_3d location)
 {
   double       x, y, z;
   int          partialX, partialZ;
   double       fractionX, fractionZ;
   double[][]   triCoords = new double[3][3];
   double[][]   vectors = new double[2][3];
   double[]     tempNormal = new double[3];
```

```
   double[]     rotationVector = new double[3];
   double[]     upVector = new double[3];
   SFRotation   outRotation;
   double       rotationAngle;
   int          index;
   MFVec3f      coord_point;
   float[]      coordinates;

   Node coord;
   coord          =
(Node)((SFNode)getField("coord")).getValue();

   // Check if this terrain grid was built using UTM
   if(geo_system.startsWith("UTM"))
   {
     // the location has to be converted to UTM also to
calculate elevation
     Utm_Coord_3d newLocation = new Utm_Coord_3d();
     Gdc_To_Utm_Converter.Init();
     Gdc_To_Utm_Converter.Convert(location, newLocation);
     x = newLocation.x;
     z = newLocation.y;
   }
   else
   {
     // This grid was built in GDC, so no conversion is
necessary
     x = location.latitude;
     z = location.longitude;
   }

   // Calculating the orientation requires using the
coordinates array that was
   //  built by buildCoordinateSet(height).  Here it is
retrieved.
   coord_point = (MFVec3f)coord.getExposedField("point");
   // Each coordinate takes 3 float values (x, y, z)
   coordinates = new float[coord_point.getSize() * 3];
   coord_point.getValue(coordinates);

   // I am assuming that every object created uses the
standard VRML convention
   //  of up being the Y axis.  GeoVRML up does not matter
here.
   upVector[0] = 0.0;
   upVector[1] = 1.0;
   upVector[2] = 0.0;
```

```
   //  This gives the number of postings to travel in the x
and z directions to
   //   reach the lower left corner - not the actual x and
z coordinate values.
   //   and the value is always truncated (rounded down).
   if(geo_system.startsWith("UTM"))
   {
     partialX = (int)((x - utmSouthWest.x)/xInc);
     partialZ = (int)((z - utmSouthWest.y)/zInc);
   }
   else
   {
     partialX = (int)((x - gdcSouthWest.latitude)/xInc);
     partialZ = (int)((z - gdcSouthWest.longitude)/zInc);
   }
   // Next, get the fraction of one space in both the X and
Z directions left over from
   //  the partials found above.  These are used to
determine which half of the square
   //  this coordinate falls in (lower left half or upper
right).  This is needed to
   //  determine which point is the third point and what
order to grab the second
   //  and third point in.
   if(geo_system.startsWith("UTM"))
   {
     fractionX = ((x - utmSouthWest.x)/xInc) - partialX;
     fractionZ = ((z - utmSouthWest.y)/zInc) - partialZ;
   }
   else
   {
     fractionX = ((x - gdcSouthWest.latitude)/xInc) -
partialX;
     fractionZ = ((z - gdcSouthWest.longitude)/zInc) -
partialZ;
   }

   // The terrain data is received as a 1 dimensional array
of height values.
   //  The array can be thought of as a checkerboard
pattern with height values
   //  coinciding with every corner of every square.  The
terrain is drawn by
   //  breaking up the checkerboard into triangles.  This
code uses the following
   //  pattern to break up the squares:
   //     +---+
```

```
   //     |  /|
   //     | / |
   //     |/  |
   //     +---+
   // Extract the three coordinates of the triangle that
contains the given location.
   //  each of these coordinates will have the geoOrigin
added to it to bring the
   //  point back into GCC space from screen space.
   // The first coordinate is always the lower left corner.
   index = (partialX + partialZ*xDimension)*3;  // index
now points at lower left corner
   triCoords[0][0] = coordinates[index] + gccGeoOrigin.x;
   triCoords[0][1] = coordinates[index+1] + gccGeoOrigin.y;
   triCoords[0][2] = coordinates[index+2] + gccGeoOrigin.z;

   // FractionX and fractionZ are used to determine which
half of the square
   //  this coordinate is in (the lower right or upper
left). If fractionX is
   //  greater than fractionZ, then this coordinate is in
the lower right half.
   if(fractionX > fractionZ)
   {
     // Get the lower right corner of the square for
coordinate 2
     index = ((partialX + 1) + partialZ*xDimension)*3;
     triCoords[1][0] = coordinates[index] + gccGeoOrigin.x;
     triCoords[1][1] = coordinates[index+1] +
gccGeoOrigin.y;
     triCoords[1][2] = coordinates[index+2] +
gccGeoOrigin.z;
     // Get the upper right corner of the square for
coordinate 3
     index = ((partialX + 1) + (partialZ +
1)*xDimension)*3;
     triCoords[2][0] = coordinates[index] + gccGeoOrigin.x;
     triCoords[2][1] = coordinates[index+1] +
gccGeoOrigin.y;
     triCoords[2][2] = coordinates[index+2] +
gccGeoOrigin.z;
   }
   else
   {
     // Get the upper right corner of the square for
coordinate 2
```

```
     index = ((partialX + 1) + (partialZ +
1)*xDimension)*3;
     triCoords[1][0] = coordinates[index] + gccGeoOrigin.x;
     triCoords[1][1] = coordinates[index+1] +
gccGeoOrigin.y;
     triCoords[1][2] = coordinates[index+2] +
gccGeoOrigin.z;
     // Get the upper left corner of the square for
coordinate 3
     index = (partialX + (partialZ + 1)*xDimension)*3;
     triCoords[2][0] = coordinates[index] + gccGeoOrigin.x;
     triCoords[2][1] = coordinates[index+1] +
gccGeoOrigin.y;
     triCoords[2][2] = coordinates[index+2] +
gccGeoOrigin.z;
   }

   // Get two vectors from the coordinates of the plane.
It is important to
   //  get the vectors in the order shown or you could get
a tangent normal pointing
   //  directly into the polygon instead of directly out of
the polygon
   for(int i = 0; i <= 2; i++)
   {
     vectors[0][i] = triCoords[0][i] - triCoords[1][i];
     vectors[1][i] = triCoords[0][i] - triCoords[2][i];
   }

   // Next, determine the normal using a cross product:
   //                                      |      i
j              k      |
   //  normal = vector[0] x vector[1] = |vectors[0][0]
vectors[0][1]  vectors[0][2]|
   //                                      |vectors[1][0]
vectors[1][1]  vectors[1][2]|
   tempNormal[0] = vectors[0][1]*vectors[1][2] -
vectors[0][2]*vectors[1][1];
   tempNormal[1] = (vectors[0][0]*vectors[1][2] -
vectors[0][2]*vectors[1][0])*(-1);
   tempNormal[2] = vectors[0][0]*vectors[1][1] -
vectors[0][1]*vectors[1][0];

   // To calculate the vector we must rotate around to make
an object coincide
   //  with the normal we have to take the cross product of
the normal vector
```

```
   //  and the vector that represents the up direction for
the object that is
   //  to be rotated.
   rotationVector[0] = upVector[1]*tempNormal[2] -
upVector[2]*tempNormal[1];
   rotationVector[1] = (-1)*(upVector[0]*tempNormal[2] -
upVector[2]*tempNormal[0]);
   rotationVector[2] = upVector[0]*tempNormal[1] -
upVector[1]*tempNormal[0];
   // Normalize the rotationVector so that the x, y, and z
values will definately
   //  fit into float variables
   double rotationMagnitude =
Math.sqrt(rotationVector[0]*rotationVector[0] +
       rotationVector[1]*rotationVector[1] +
rotationVector[2]*rotationVector[2]);
   rotationVector[0] = rotationVector[0]/rotationMagnitude;
   rotationVector[1] = rotationVector[1]/rotationMagnitude;
   rotationVector[2] = rotationVector[2]/rotationMagnitude;
   // We also need the angle between the upVector and
normal vector so that we
   //  know how far to rotate the object to coincide with
the normal vector.
   //  The cosine of this angle is equal to the dot product
of the upVector
   //  and normal vector.
   double dotProduct = upVector[0]*tempNormal[0] +
upVector[1]*tempNormal[1] +
                       upVector[2]*tempNormal[2];
   double upMagnitude = Math.sqrt(upVector[0]*upVector[0] +
upVector[1]*upVector[1] +
                                  upVector[2]*upVector[2]);
   double normalMagnitude =
Math.sqrt(tempNormal[0]*tempNormal[0] +
tempNormal[1]*tempNormal[1]
                                       +
tempNormal[2]*tempNormal[2]);
   rotationAngle =
Math.acos(dotProduct/(upMagnitude*normalMagnitude));
   // We now have what we need to build the rotation vector
that the target object
   //  will use to rotate itself to coincide with the
normal
   outRotation = new SFRotation(new
Double(rotationVector[0]).floatValue(),
                         new
Double(rotationVector[1]).floatValue(),
```

```
                        new
Double(rotationVector[2]).floatValue(),
                        new
Double(rotationAngle).floatValue());
   return outRotation;
 }

 public Gdc_Coord_3d getElevation(Gdc_Coord_3d location)
 {
   double       x, y, z;
   int          partialX, partialZ;
   double       fractionX, fractionZ;
   double[][]   triCoords = new double[3][3];
   double[][]   vectors = new double[2][3];
   double[]     tempNormal = new double[3];
   double[]     upVector = new double[3];
   int          index;
   Gdc_Coord_3d outLocation = new Gdc_Coord_3d();
   MFFloat      height;
   float[]      heightArray;
   double       xOrigin, zOrigin;

   if(geo_system.startsWith("UTM"))
   {
     // the location has to be converted to UTM also to
calculate elevation
     Utm_Coord_3d newLocation = new Utm_Coord_3d();
     Gdc_To_Utm_Converter.Init();
     Gdc_To_Utm_Converter.Convert(location, newLocation);
     x = newLocation.x;
     z = newLocation.y;
     xOrigin = utmSouthWest.x;
     zOrigin = utmSouthWest.y;
   }
   else
  {
     // This grid was built in GDC, so no conversion is
necessary
     x = location.latitude;
     z = location.longitude;
     xOrigin = gdcSouthWest.latitude;
     zOrigin = gdcSouthWest.longitude;
   }

   height = (MFFloat)getField("height");
   heightArray = new float[height.getSize()];
   height.getValue(heightArray);
```

```
   // Get the partial values - these are the nearest whole
number coordinates
   //  past zero to locate the lower left corner of the
square this coordinate is in.
   //  Note: this gives the number of postings to travel in
the x and y directions to
   //   reach the lower left corner - not the actual x and
y coordinate values.
   partialX = (int)((x - xOrigin)/xInc);
   partialZ = (int)((z - zOrigin)/zInc);

   // Next, get the fraction of one space in both the X and
Z directions left over from
   //  the partials found above.  These are used to
determine which half of the square
   //  this coordinate falls in (lower left half or upper
right).  This is needed to
   //  determine which point is the third point and what
order to grab the second
   //  and third point in.
   fractionX = ((x - xOrigin)/xInc) - partialX;
   fractionZ = ((z - zOrigin)/zInc) - partialZ;

   // We need to get 3 coordinates so we can define a plane
and
   //  determine the elevation at a specific point
   // The first coordinate is the lower left corner of this
grid box
   triCoords[0][0] = (partialX * xInc) + xOrigin;
   triCoords[0][1] = heightArray[(partialZ * zDimension +
partialX)];
   triCoords[0][2] = (partialZ * zInc) + zOrigin;
   // The fractionX and fractionZ are used to determine
which half of the square
   //  this coordinate is in (the lower right or upper
left).
   //  If fractionX is greater than fractionZ, then this
coordinate is in the
   //  lower right half of the square.  This means we need
the lower right corner
   //  and the upper right triangle for the 2nd and 3rd
coordinates.
   if(fractionX > fractionZ)
   {
     // Get the lower right corner of the square for
coordinate 2
```

```
     triCoords[1][0] = ((partialX + 1) * xInc) + xOrigin;
     triCoords[1][1] = heightArray[partialZ*zDimension +
(partialX + 1)];
     triCoords[1][2] = (partialZ * zInc) + zOrigin;
     // Get the upper right corner of the square for
coordinate 3
     triCoords[2][0] = ((partialX + 1) * xInc) + xOrigin;
     triCoords[2][1] = heightArray[(partialZ +
1)*zDimension + (partialX + 1)];
     triCoords[2][2] = ((partialZ + 1) * zInc) + zOrigin;
  }
   // The else clause means that we are in the upper left
half of the square
   //  and need the upper right corner and upper left
corner.
   else
   {
     // Get the upper right corner of the square for
coordinate 2
     triCoords[1][0] = ((partialX + 1) * xInc) + xOrigin;
     triCoords[1][1] = heightArray[(partialZ +
1)*zDimension + (partialX + 1)];
     triCoords[1][2] = ((partialZ + 1) * zInc) + zOrigin;
     // Get the upper left corner of the square for
coordinate 3
     triCoords[2][0] = (partialX * xInc) + xOrigin;
     triCoords[2][1] = heightArray[(partialZ +
1)*zDimension + partialX];
     triCoords[2][2] = ((partialZ + 1) * zInc) + zOrigin;
   }

   // The 3 coordinates just found are what we need to
determine the elevation at the
   //  specified point and the normal for the location.
   // First, get two vectors from the coordinates of the
plane.  It is important to
   //  get the vectors in the order shown or you could get
a tangent normal pointing
   //  directly into the polygon instead of directly out of
the polygon
   for(int i = 0; i <= 2; i++)
   {
     vectors[1][i] = triCoords[0][i] - triCoords[2][i];
     vectors[0][i] = triCoords[0][i] - triCoords[1][i];
   }

   // Next, determine the normal using a cross product:
```

```
   //                                      |      i
j              k      |
   //  normal = vector[0] x vector[1] = |vectors[0][0]
vectors[0][1]  vectors[0][2]|
   //                                      |vectors[1][0]
vectors[1][1]  vectors[1][2]|
   tempNormal[0] = vectors[0][1]*vectors[1][2] -
vectors[0][2]*vectors[1][1];
   tempNormal[1] = (vectors[0][0]*vectors[1][2] -
vectors[0][2]*vectors[1][0])*(-1);
   tempNormal[2] = vectors[0][0]*vectors[1][1] -
vectors[0][1]*vectors[1][0];

   // Finally, calculate the height for the original
coordinate using the equation:
   //
   //   0 = normal[0](x - coordinates[0][0]) + normal[1](y
- coordinates[0][1]) +
   //       normal[2](z - coordinates[0][2]
   //
   y = ((-1)*tempNormal[0]*(x - triCoords[0][0]) -
tempNormal[2]*(z - triCoords[0][2]))
       / tempNormal[1] + triCoords[0][1];

   // We now have the calculated elevation which can be
combined with the lat
   //  and long values passed in to get the new coordinate
with elevation equal
   //  to the terrain.
   outLocation.elevation = y;
   outLocation.latitude = location.latitude;
   outLocation.longitude = location.longitude;

   return outLocation;
 }
 // ************* End CPT Brian Hittner Addition Sep 2003
**************
}

// EOF: GeoElevationGrid.java
```

C. GEOLOCATION3 NODE

This class is the interface for objects that need to use GeoTerrainGrids. This class allows the user to decide whether to have GeoTerrainGrids automatically set the elevation of the object to surface level and whether to have the orientation of the object coincide with the normal of the terrain. Otherwise, the node operates exactly as its predecessors GeoLocation and GeoLocation2. This code has the same license as the GeoTerrainGrid does as it is also originally written by SRI.

```
//
// Filename: GeoLocation3.java
//
// Authors:
//   Martin Reddy, SRI International - 21 August 1999
//   John Brecht, SRI Internation - 31 March 2000
//   CPT Brian Hittner, US Army, Naval Postgraduate School
student - Sep 2003
//
// Purpose:
//   This class implements a new Transform node for VRML.
It allows you
//   to take any arbitrary VRML context and geo-reference
it, i.e. place
//   it at a specific point on the planet.
//
//   This code requires access to the GeoTransform Java
package, included
//   as part of the GeoVRML source code distribution.
//
// License:
//   The contents of this file are subject to GeoVRML
Public License
//   Version 1.0 (the "License"); you may not use this file
except in
//   compliance with the License. You may obtain a copy of
the License at
//   http://www.geovrml.org/1.0/license/.
//
//   Software distributed under the License is distributed
on an "AS
//   IS" basis, WITHOUT WARRANTY OF ANY KIND, either
express or
//   implied. See the License for the specific language
governing
//   rights and limitations under the License.
//
```

```
//   Portions are Copyright (c) SRI International, 2000.
//
// Revision:
//  $Id: GeoLocation.java,v 1.1.1.1 2000/06/15 16:49:27
reddy Exp $
//
//  Martin Reddy (21 Aug 1999) - initial version
//  John Brecht (31 Mar 2000) - support set_geoCoords
eventIn
//  Brian Hittner (Sep 2003) - works with GeoManager and
added support
//                          for autoElevation and
autoSurfaceOrientation booleans
//

import java.lang.*;
import vrml.*;
import vrml.field.*;
import vrml.node.*;
import geotransform.coords.*;
import org.web3d.geovrml.GeoVRML;
// ****************** Start CPT Brian Hittner change
********************
import java.util.*;
// ******************  End CPT Brian Hittner change
********************

public class GeoLocation3 extends Script {

  MFString geoSystem;
  SFNode geoOrigin;
  GeoVRML geovrml;
  Node transform;
  boolean debug;
  SFString geoCoords_changed;
// ****************** Start CPT Brian Hittner change
********************
  boolean autoElevation, autoSurfaceOrientation;
  GeoManager manager;
  SFBool autoElevation_changed,
autoSurfaceOrientation_changed;
// ******************  End CPT Brian Hittner change
********************


  // process the set_geoCoords eventIn by calling
updateGeoCoords()
```

```
  // and produce the geoCoords_changed eventOut.


  public void processEvent( Event e ) {
    String name = e.getName();
    if ( debug ) System.out.println( "Event received: " +
name );
    if ( name.equals( "set_geoCoords" ) ) {
      ConstSFString csfstring = (ConstSFString)
e.getValue();
      SFString sfstring = new SFString(
csfstring.getValue() );
      updateGeoCoords(sfstring);
      geoCoords_changed.setValue(sfstring);
    }
// ***************** Start CPT Brian Hittner change
*******************
    if(name.equals("set_autoElevation"))
    {
      ConstSFBool csfbool = (ConstSFBool)e.getValue();
      autoElevation = csfbool.getValue();
      ConstSFString csfstring = (ConstSFString)
e.getValue();
      SFString sfstring = new SFString(
csfstring.getValue() );
      updateGeoCoords(sfstring);
      autoElevation_changed.setValue(csfbool);
    }
    if(name.equals("set_autoSurfaceOrientation"))
    {
      ConstSFBool csfbool2 = (ConstSFBool)e.getValue();
      autoSurfaceOrientation = csfbool2.getValue();
      ConstSFString csfstring = (ConstSFString)
e.getValue();
      SFString sfstring = new SFString(
csfstring.getValue() );
      updateGeoCoords(sfstring);
      autoSurfaceOrientation_changed.setValue(csfbool2);
    }
// *****************  End CPT Brian Hittner change
*******************
  }

  // The initialize method is called when the Node is first
loaded.
  // Here we grab copies of any necessary
fields/eventIn/eventOuts
```

```
  // and do the coordinate transformation in order to find
the
  // correct geolocation for the transform's childrens.
  public void initialize() {
    // Take copies of all the fields for this node
    geoOrigin   = (SFNode) getField( "geoOrigin" );
    geoSystem = (MFString) getField( "geoSystem" );
    SFString geoCoords = (SFString) getField( "geoCoords"
);
    transform     = (Node) ((SFNode) getField( "transform"
)).getValue();
    debug      = ((SFBool) getField( "debug" )).getValue();
    geoCoords_changed = (SFString) getEventOut(
"geoCoords_changed" );

// ***************** Start CPT Brian Hittner change
*******************
    autoElevation =
((SFBool)getField("autoElevation")).getValue();
    autoSurfaceOrientation =
((SFBool)getField("autoSurfaceOrientation")).getValue();
    autoElevation_changed =
(SFBool)getEventOut("autoElevation_changed");
    autoSurfaceOrientation_changed =

(SFBool)getEventOut("autoSurfaceOrientation_changed");
    manager = GeoManager.getGeoManager();
// *****************  End CPT Brian Hittner change
*******************

    if ( debug ) System.out.println( "GeoLocation:" );

    // Okay, let's initialise the GeoVRML utility class
    // These classes should be installed on the user's
system and in
    // their CLASSPATH. If they are not, then we can't do
anything!
    try {
      geovrml = new GeoVRML();
    } catch ( NoClassDefFoundError e ) {
      System.out.println( "GeoTransform classes not
installed in CLASSPATH!" );
      return;
    }
    geovrml.setOrigin( geoOrigin );
    updateGeoCoords(geoCoords);
  }
```

```
    // Converts the inputed geoCoords to GCC and sets the
transform
    // of the Node appropriately.
  public void updateGeoCoords(SFString geoCoords) {
// ****************** Start CPT Brian Hittner change
********************
    Gdc_Coord_3d tempPosition = null;
    GeoTerrainGrid terrainGrid = null;
    SFRotation tempRotation;
    // If autoElevation or autoSurfaceOrientation is set,
then a GeoTerrainGrid
    //  for this location is needed
    if(autoElevation || autoSurfaceOrientation)
    {
      // The coordinate must be in GDC, so UTM coordinates
must be converted
      if(geoSystem.toString().startsWith("UTM"))
        tempPosition =
convertUtmToGdc(geoCoords.getValue());
      else
        tempPosition = parseGDC(geoCoords.getValue());
      terrainGrid = manager.getGrid(tempPosition);
      if(terrainGrid==null)
      {
        debugOut("Manager did not locate a suitable terrain
grid for: " +
                 geoCoords.toString());
        debugOut("GDC_Coord_3d: Latitude: " +
tempPosition.latitude + " Longitude: "
                 + tempPosition.longitude + " Elevation: "
+ tempPosition.elevation);
        return;
      }
      else
      {
        // When autoElevation is true, use the terrain grid
to determine elevation
        if(autoElevation)
        {
          // This call to getElevation() will fill in the
proper elevation
          tempPosition =
terrainGrid.getElevation(tempPosition);
          // Replace geoCoords current value with the new
value that has elevation
          geoCoords.setValue(gdcToString(tempPosition));
```

```
        }
      }
    }
// *****************  End CPT Brian Hittner change
********************

    // Find out the location that the user wants to
georeference to
    // This is essentially the translation vector for the
transform
    Gcc_Coord_3d gcc = geovrml.getCoord( geoCoords,
geoSystem );
    SFVec3f xform_trans = (SFVec3f)
transform.getExposedField( "translation" );
    xform_trans.setValue( (float) gcc.x, (float) gcc.y,
(float) gcc.z );
    if ( debug )
      System.out.println( "  translation = " + gcc.x +" "+
gcc.y +" "+ gcc.z );

    // Now let's work out the orientation at that location
in order
    // to maintain a view where +Y is in the direction of
gravitional
    // up for that region of the planet's surface. This
will be the
    // value of the rotation vector for the transform.
    float orient[] = new float[4];

    SFRotation xform_rot = (SFRotation)
transform.getExposedField("rotation");
// ***************** Start CPT Brian Hittner change
********************
    if(autoSurfaceOrientation)
    {
      tempRotation =
terrainGrid.getOrientation(tempPosition);
      xform_rot.setValue(tempRotation);
    }
// *****************  End CPT Brian Hittner change
********************
    else
    {
      geovrml.getLocalOrientation(gcc, orient);
      xform_rot.setValue(orient[0], orient[1], orient[2],
orient[3]);
    }
```

```
    if ( debug )
      System.out.println( "  rotation = " + orient[0] + " "
+ orient[1] +
                " " + orient[2] + " " + orient[3] + " (" +
                orient[3] * 57.29578f + " deg)" );

    // Finally, we can set the scale field of the transform
based
    // upon the global GeoVRML class scaleFactor.
    SFVec3f xform_scale = (SFVec3f)
transform.getExposedField( "scale" );
    float   scale = (float) ( 1.0 / geovrml.scaleFactor );
    xform_scale.setValue( scale, scale, scale );
  }

// ***************** Start CPT Brian Hittner change
********************
  // Takes a gdcCoordinate as a String and turns it into a
Gdc_Coord_3d
  private Gdc_Coord_3d parseGDC(String gdcCoordinate)
  {
    Gdc_Coord_3d coord;
    double[] array = new double[3];
    StringTokenizer tokenizer = new
StringTokenizer(gdcCoordinate, " ");
    for(int i = 0; i <= 2; i++)
      array[i] = new
Double(tokenizer.nextToken()).doubleValue();
    coord = new Gdc_Coord_3d(array[0], array[1], array[2]);
    return coord;
  }

  // Takes a Gdc_Coord_3d coordinate, and returns a string
(in GDC)
  private String gdcToString(Gdc_Coord_3d coordinate)
  {
    String coordinateString = new String();
    coordinateString += new
Double(coordinate.latitude).toString();
    coordinateString += " ";
    coordinateString += new
Double(coordinate.longitude).toString();
    coordinateString += " ";
    coordinateString += new
Double(coordinate.elevation).toString();
    return coordinateString;
```

```
  }

  // Warning: this routine only works if the current
geoSystemString is set
  //  to UTM with a zone included
  private Gdc_Coord_3d convertUtmToGdc(String utm)
  {
    Gcc_Coord_3d tempGcc = geovrml.getCoord(utm,
geoSystem.toString());
    float[] floatGridArray = new float[3];
    floatGridArray[0] = new Double(tempGcc.x).floatValue();
    floatGridArray[1] = new Double(tempGcc.y).floatValue();
    floatGridArray[2] = new Double(tempGcc.z).floatValue();
    String tempLocation = geovrml.geoCoord(floatGridArray,
"GD");
    // tempOrigin is now in GD coordinate space (lat/long)
as a String
    Gdc_Coord_3d tempGdc = parseGDC(tempLocation);
    return tempGdc;
  }

  public void debugOut(String message)
  {
    if(debug)
      System.out.println(message);
  }
// *****************  End CPT Brian Hittner change
********************
}

// EOF: GeoLocation.java
```

D. EXAMPLE OF GEOTERRAINGRID

This code is shown in VRML format and declares a GeoTerrainGrid. This code does not display anything by itself because it does not have any viewpoint defined. The purpose of this code is to be used as part of another X3D or VRML program. The majority of the data for the terrain was removed, though, because it just generated pages of numbers. For the full version of this program with the data intact, check the SAVAGE website. There is an indexed line set in this code that is not needed for the code to work. It was added simply to outline the terrain which makes each individual terrain grid much more visible when rendered as part of a group of terrain grids

```
#VRML V2.0 utf8
# X3D-to-VRML-97 XSL translation autogenerated by
X3dToVrml97.xsl
#
http://www.web3D.org/TaskGroups/x3d/translation/X3dToVrml97
.xsl

# [X3D] VRML V3.0 utf8
EXTERNPROTO GeoCoordinate [
  field  SFNode    geoOrigin    # NULL
  field  MFString  geoSystem    # [ "GDC" ]
  field  MFString  point        # []
] [
    "GeoVRML/1.1/protos/GeoCoordinate.wrl#GeoCoordinate"

"../../GeoVRML/1.1/protos/GeoCoordinate.wrl#GeoCoordinate"
    "C:/Program
Files/GeoVRML/1.1/protos/GeoCoordinate.wrl#GeoCoordinate"
    "file:///C|/Program
Files/GeoVRML/1.1/protos/GeoCoordinate.wrl#GeoCoordinate"

"urn:web3d:geovrml:1.0/protos/GeoCoordinate.wrl#GeoCoordina
te"

"http://www.geovrml.org/1.0/protos/GeoCoordinate.wrl#GeoCoo
rdinate"
  ]
EXTERNPROTO GeoTerrainGrid [
  field        SFNode     geoOrigin         #NULL
  field        MFString   geoSystem         #["GD" "WE"]
  field        SFString   geoGridOrigin     #"0 0 0"
  field        SFInt32    xDimension        #0       # [0,)
```

```
  field        SFString   xSpacing          #"1.0"   # (0,)
  field        SFInt32    zDimension        #0       # [0,)
  field        SFString   zSpacing          #"1.0"   # (0,)
  field        MFFloat    height            #[]      # (-,)
  eventIn      MFFloat    set_height
  field        SFFloat    yScale            #1.0
  eventIn      SFFloat    set_yScale
  exposedField SFNode     color             #NULL
  exposedField SFNode     texCoord
#TextureCoordinate {}
  exposedField SFNode     normal            #NULL
  field        SFBool     normalPerVertex   #TRUE
  field        SFBool     ccw               #TRUE
  field        SFBool     colorPerVertex    #TRUE
  field        SFFloat    creaseAngle       #0       # [0,]
  field        SFBool     solid             #TRUE
] [
     "GeoTerrainGrid.wrl#GeoTerrainGrid"
  ]
EXTERNPROTO GeoMetadata [
  exposedField MFString url        # []
  exposedField MFString summary    # []
  exposedField MFNode   data       # []
] [
    "../../GeoVRML/1.1/protos/GeoMetadata.wrl#GeoMetadata"
    "GeoVRML/1.1/protos/GeoMetadata.wrl#GeoMetadata"
    "C:/Program
Files/GeoVRML/1.1/protos/GeoMetadata.wrl#GeoMetadata"
    "file:///C|/Program
Files/GeoVRML/1.1/protos/GeoMetadata.wrl#GeoMetadata"

"urn:web3d:geovrml:1.0/protos/GeoMetadata.wrl#GeoMetadata"

"http://www.geovrml.org/1.0/protos/GeoMetadata.wrl#GeoMetad
ata"
  ]
EXTERNPROTO GeoOrigin [
 exposedField  MFString  geoSystem    # [ "GDC" ]
 exposedField  SFString  geoCoords    # ""
 field         SFBool    rotateYUp    # FALSE
] [
    "GeoVRML/1.1/protos/GeoOrigin.wrl#GeoOrigin"
    "../../GeoVRML/1.1/protos/GeoOrigin.wrl#GeoOrigin"
    "C:/Program
Files/GeoVRML/1.1/protos/GeoOrigin.wrl#GeoOrigin"
    "file:///C|/Program
Files/GeoVRML/1.1/protos/GeoOrigin.wrl#GeoOrigin"
```

```
    "urn:web3d:geovrml:1.0/protos/GeoOrigin.wrl#GeoOrigin"

"http://www.geovrml.org/1.0/protos/GeoOrigin.wrl#GeoOrigin"
  ]
# [Scene]

GeoMetadata {
  summary [ "DTED2, N290E520" ]
  url [ "N290E520DTED2.wrl" ]
}
DEF ORIGIN GeoOrigin {
  geoCoords "29.0 52.0 0.0"
}
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.4 0.6 0.3
      emissiveColor 0 0.05 0
    }
  }
  geometry GeoTerrainGrid {
    geoOrigin USE ORIGIN
    creaseAngle .785
    geoGridOrigin "29.71186440677966 52.6271186440678 0"
    geoSystem [ "GDC" ]
    height [ 2013 2012 2011 2009 2007 2007 2008 2010 2013
2015 2017 2022
                    --- Data Removed for Brevity ---
1972 1978 1986 1995 2000 1999 1993 1981 1970 1957 1946 1934
1921 ]
    xDimension 61
    xSpacing "2.824074074074074E-4"
    zDimension 61
    zSpacing "2.824074074074074E-4"
  }
}
Shape {
  appearance Appearance {
    material Material {
      emissiveColor 0.8 0.8 0.8
    }
  }
  geometry DEF LINESET IndexedLineSet {
    coordIndex [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
                    --- Data Removed for Brevity ---
232 233 234 235 236 237 238 239 240 241 242 243 -1 -1 ]
```

```
    coord GeoCoordinate {
      geoOrigin USE ORIGIN
      geoSystem [ "GDC" ]
      point [ "29.7118644 52.6271186 2013.0 29.7121468
52.6271186 2004.0
                    --- Data Removed for Brevity ---
52.64406308851225 1921.0" ]
    }
  }
}
```

E. MULTIPLE GEOTERRAINGRIDS AND A GEOLOCATION3 EXAMPLE

This is an example file that brings everything together. Multiple GeoTerrainGrids are created and a tank is driven across the grids which is automatically positioned on the terrain and oriented to it. This program requires seventeen other files which are not listed. However, the entire demo is posted on the SAVAGE web site. This program is an excellent example of how current GeoVRML structures such as a GeoPositionInterpolator will work with the new nodes created in this thesis. The tank in the program travels seamlessly through the GeoTerrainGrids in the scene.

```
#VRML V2.0 utf8
# X3D-to-VRML-97 XSL translation autogenerated by
X3dToVrml97.xsl
#
http://www.web3D.org/TaskGroups/x3d/translation/X3dToVrml97
.xsl

# [X3D] VRML V3.0 utf8

# [head]
EXTERNPROTO GeoLocation3 [
  field    SFNode   geoOrigin             #NULL
  field    MFString geoSystem             #[ "GDC" ]
  field    SFString geoCoords             #""
  field    MFNode   children              #[]
  field      SFBool   autoElevation         #FALSE
  field      SFBool   autoSurfaceOrientation #FALSE
  field    SFBool   debug                   #FALSE
  eventIn  SFString set_geoCoords
  eventOut SFString geoCoords_changed
  eventIn  SFBool   set_autoElevation
  eventOut SFBool   autoElevation_changed
  eventIn  SFBool   set_autoSurfaceOrientation
  eventOut SFBool   autoSurfaceOrientation_changed
] [
     "GeoLocation3.wrl#GeoLocation3"
]

EXTERNPROTO GeoOrigin [
 exposedField  MFString  geoSystem    # [ "GDC" ]
 exposedField  SFString  geoCoords    # ""
 field         SFBool    rotateYUp    # FALSE
] [
```

```
    "GeoVRML/1.1/protos/GeoOrigin.wrl#GeoOrigin"
    "../../GeoVRML/1.1/protos/GeoOrigin.wrl#GeoOrigin"
    "C:/Program
Files/GeoVRML/1.1/protos/GeoOrigin.wrl#GeoOrigin"
    "file:///C|/Program
Files/GeoVRML/1.1/protos/GeoOrigin.wrl#GeoOrigin"
    "urn:web3d:geovrml:1.0/protos/GeoOrigin.wrl#GeoOrigin"

"http://www.geovrml.org/1.0/protos/GeoOrigin.wrl#GeoOrigin"
  ]

EXTERNPROTO GeoViewpoint [
 field         SFNode      geoOrigin        # NULL
 field         MFString    geoSystem        # ["GDC"]
 field         SFString    position         # "0 0 100000"
 field         SFRotation  orientation      # 0 0 1 0
 exposedField  SFFloat     fieldOfView      # 0.785398
 exposedField  SFBool      jump             # TRUE
 exposedField  MFString    navType          #
["EXAMINE","ANY"]
 exposedField  SFBool      headlight        # TRUE
 field         SFString    description      # ""
 field         SFFloat     speed            # 1.0
 eventIn       SFString    set_position
 eventIn       SFString    set_orientation
 eventIn       SFBool      set_bind
 eventOut      SFTime      bindTime
 eventOut      SFBool      isBound
] [
    "GeoVRML/1.0/protos/GeoViewpoint.wrl#GeoViewpoint"

"../../GeoVRML/1.0/protos/GeoViewpoint.wrl#GeoViewpoint"
    "C:/Program
Files/GeoVRML/1.0/protos/GeoViewpoint.wrl#GeoViewpoint"
    "file:///C|/Program
Files/GeoVRML/1.0/protos/GeoViewpoint.wrl#GeoViewpoint"

"urn:web3d:geovrml:1.0/protos/GeoViewpoint.wrl#GeoViewpoint
"

"http://www.geovrml.org/1.0/protos/GeoViewpoint.wrl#GeoView
point"
  ]

EXTERNPROTO    GeoPositionInterpolator [
 eventIn  SFFloat   set_fraction
 field    SFNode    geoOrigin
```

```
 field    MFString  geoSystem
 field    MFFloat   key
 field    MFString  keyValue
 eventOut SFVec3f   value_changed
 eventOut SFString  geovalue_changed
] [

"GeoVRML/1.0/protos/GeoPositionInterpolator.wrl#GeoPosition
Interpolator"

"../../GeoVRML/1.0/protos/GeoPositionInterpolator.wrl#GeoPo
sitionInterpolator"
    "C:/Program
Files/GeoVRML/1.0/protos/GeoPositionInterpolator.wrl#GeoPos
itionInterpolator"
    "file:///C|/Program
Files/GeoVRML/1.0/protos/GeoPositionInterpolator.wrl#GeoPos
itionInterpolator"

"urn:web3d:geovrml:1.0/protos/GeoPositionInterpolator.wrl#G
eoPositionInterpolator"

"http://www.geovrml.org/1.0/protos/GeoPositionInterpolator.
wrl#GeoPositionInterpolator"
]

# [Scene]

NavigationInfo {
  speed 5000
}
DEF ORIGIN GeoOrigin {
  geoCoords "29.0 52.0 0.0"
}
DEF ViewPoint GeoViewpoint {
  geoOrigin USE ORIGIN
  geoSystem [ "GD" "WE" ]
  orientation 1.0 0.0 0.0 -0.5
  position "29.68 52.66 4000"
}
DEF Top Group {
  children [
      Inline {
        url [

     #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2942E5237DTED2.wrl"
```

```
                "N2942E5237DTED2.wrl"
            ]
        }
        Inline {
          url [

      #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2942E5238DTED2.wrl"
                "N2942E5238DTED2.wrl"
                ]
        }
        Inline {
          url [

      #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2942E5239DTED2.wrl"
                "N2942E5239DTED2.wrl"
                ]
        }
        Inline {
          url [

      #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2942E5240DTED2.wrl"
                "N2942E5240DTED2.wrl"
                ]
        }
        Inline {
          url [

      #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2943E5237DTED2.wrl"
                "N2943E5237DTED2.wrl"
                ]
        }
        Inline {
          url [

      #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2943E5238DTED2.wrl"
                "N2943E5238DTED2.wrl"
                ]
        }
        Inline {
          url [
```

```
     #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2943E5239DTED2.wrl"
                "N2943E5239DTED2.wrl"
                ]
      }
      Inline {
        url [

     #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2943E5240DTED2.wrl"
                "N2943E5240DTED2.wrl"
                ]
      }
      Inline {
        url [

     #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2944E5237DTED2.wrl"
                "N2944E5237DTED2.wrl"
                ]
      }
      Inline {
        url [

     #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2944E5238DTED2.wrl"
                "N2944E5238DTED2.wrl"
                ]
      }
      Inline {
        url [

     #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2944E5239DTED2.wrl"
                "N2944E5239DTED2.wrl"
                ]
      }
      Inline {
        url [

     #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2944E5240DTED2.wrl"
                "N2944E5240DTED2.wrl"
                ]
      }
      Inline {
```

```
        url [

     #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2945E5237DTED2.wrl"
                "N2945E5237DTED2.wrl"
                ]
       }
       Inline {
         url [

     #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2945E5238DTED2.wrl"
                "N2945E5238DTED2.wrl"
                ]
       }
       Inline {
         url [

     #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2945E5239DTED2.wrl"
                "N2945E5239DTED2.wrl"
                ]
       }
       Inline {
         url [

     #"http://localhost:9090/GEODATA/DTED/VRML/N29E52/Terra
in/N2945E5240DTED2.wrl"
                "N2945E5240DTED2.wrl"
                ]
       }
   ]
}

DEF  UNITLOC GeoLocation3     {
     geoOrigin   USE ORIGIN
     geoSystem   [ "GD" "WE" ]
     geoCoords   "29.7118644 52.6271186 0"
     children [
          Transform {
               rotation 0.0 1.0 0.0 0.7854
               scale    10.0 10.0 10.0
               translation    0.0 15.0 0.0
               children [
                    Inline {
                         url  ["M1A1.wrl"]
                    }
```

```
                ]
            }
        ]
        autoElevation TRUE
        autoSurfaceOrientation TRUE
        debug       TRUE
}

DEF  Interpolator GeoPositionInterpolator {
        geoOrigin USE ORIGIN
        geoSystem [ "GD" "WE" ]
        key [0.0 0.99]
       keyValue ["29.711865 52.62712 0.0" "29.762711 52.677966
0.0"]
}

DEF  Clock TimeSensor {
        cycleInterval 100.0
        loop TRUE
}

ROUTE Clock.fraction_changed TO
        Interpolator.set_fraction
ROUTE Interpolator.geovalue_changed     TO
UNITLOC.set_geoCoords
```

LIST OF REFERENCES

Ames, Andrea L., Nadeau, David R., and Moreland, John L., *VRML 2.0 Sourcebook*, Wiley, 1997.

Brutzman, Don, Scenario Authoring and Visualization for Advanced Graphical Environments (SAVAGE Website), web.nps.navy.mil/~brutzman/Savage/contents.html.

Clynch, James R., "Coordinates," Paper for Naval Postgraduate School, 2002.

Clynch, James R., "Coordinates and Maps," [http://www.oc.nps.navy.mil/oc2902w/c_mtutor/index.html], March 2003.

Dahmann, Judith S., and others, "The DoD High Level Architecture: an Update," Proceedings of the 1998 Winter Simulation Conference, 1998.

Department of Defense, Performance Specification, MIL-PRF-89020A, Performance Specification Digital Terrain Elevation Data (DTED), 19 April 1996.

Duchaineau, Mark, and others, "ROAMing Terrain: Real-time Optimally Adapting Meshes," Los Alamos National Laboratory, 1999.

Dutch, Stever, "The Universal Transverse Mercator System," [www.uwgb.edu/dutchs/FieldMethods/UTMSystem.htm], August 2003.

FM 3-25-26 Map Reading and Land Navigation, 20 July 2001

FM 3-34-230 Topographic Operations, August 2000.

Neushul, James D., "Interoperability, Data Control and Battlespace Visualization Using XML, XSLT and X3D," Naval Postgraduate School, September 2003.

Hunter, David., and others, *Beginning XML, 2nd Edition*, Wrox Press, 2001.

Lindstrom, P., and Pascucci, V., "Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization," Lawrence Livermore National Laboratory, 2002.

Polack, Trent, *3D Terrain Programming*, Premier Press, 2003.

Serin, Ekrem, "Design and Test of the Cross-Format Schema Protocol (XFSP) for Networked Virtual Environments," Naval PostGraduate School, March 2003.

Shanmugan, B. and Pullen, J.M., “Software Design for Implementation of the Selectively Reliable Multicast protocol,” Proceedings of the IEEE Distributed Simulation and Real Time Applications Workshop (DS-RT ’02), Dallas, TX, October 2002.

Tamas, Rajacsics, “Real-Time Visualization of Detailed Terrain,” Budapest, 2003.

Web 3D Consortium, GeoVRML Specification, http://www.geovrml.org/geotransform/.

Web 3D Consortium, Specification Version 1.1, GeoVRML Specification, 14 July 2002.

Web 3D Consortium, Working Draft Specification, *Extensible 3D Graphics Specification*, 1999.

Xj3D Open Source VRML/X3D Toolkit website, [http://www.web3d.org/TaskGroups/source/xj3d.html].

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Don Brutzman
   Naval Postgraduate School
   Monterey, California

4. Major Nick Wittwer
   Naval Postgraduate School
   Monterey, California